Unidad 20

Descargar estos apunte en pdf o html

Índice

Programación funcional

Introducción

La **programación funcional** es un **paradigma de programación declarativa** basado en el uso de **funciones matemáticas**, en contraste con la programación **imperativa**, que enfatiza los cambios de estado mediante la mutación de variables.

Existen lenguaje de programación funcionales puros como por ejemplo **Haskell** y **Clojure**. Sin embargo, el paradigma funcional ha ido cogiendo cada vez más peso en la programación moderna. Tanto es así, que lenguajes imperativos y orientados a objetos tradicionales como **C++**, **Java**, **C#** o **PHP** han incluido características de la programación funcional en su sintaxis o potenciando las que ya tenían lenguajes como **JavaScript**, **Python**. Además, los lenguajes multi-paradigma de nueva creación como **Scala**, **Go**, **F#** o **Kotlin** (este último en menor medida) se han diseñado para que tenga más peso la programación funcional que la orientada a objetos e imperativa.

Simplificando y a grandes rasgos podemos decir que **se basa en los siguientes pilares** matemáticos:

- Cálculo Lambda.
- Clausuras.
- Teoría de categorías y más en concreto el concepto de functor y su aplicación a las ciencias de la computación a través de:
  - Mónadas.
  - Covarianza y Contravarianza.
- Recursión o Recursividad (Formaría parte del Cálculo Lambda).

Cómo se puede observar, su bases son bastante amplias y llevaría bastante ahondar todos estos conceptos. Por esta razón, **este tema pretende ser meramente introductorio y por tanto en él solo vamos a abordar algunos conceptos a través del lenguaje C#**. Estos conceptos serán extrapolables a otros lenguajes ya que como hemos comentados todos utilizan, en mayor o medida, dicho paradigma.

Algunas definiciones importantes

Algunos de los **conceptos a destacar** en la programación funcional son:

- **Función**: En programación funcional, una **función** es un bloque de código que toma uno o más argumentos de entrada y produce un valor de salida. A diferencia de las funciones en la programación imperativa, las funciones en la programación funcional son ***'ciudadanos de***

primera clase', lo que significa que pueden ser tratadas como cualquier otro valor, como números o cadenas. Esto implica que las funciones pueden ser:
    - **Asignadas a variables**.
    - **Pasadas como argumentos a otras funciones**.
    - **Devueltas como valores desde otras funciones**.
    - Almacenadas en estructuras de datos.
    - Combinadas para crear nuevas funciones.
    - Anidadas dentro de otras funciones.
- **Funciones lambda**: Las funciones lambda son funciones anónimas que se pueden definir de manera concisa y se utilizan comúnmente para operaciones de orden superior. Se **tratan como expresiones** que pueden ser asignadas a variables o pasadas como argumentos. (Las trataremos más adelante).
- **Funciones de orden superior (HOF)**: Las funciones de orden superior son aquellas que pueden tomar otras funciones como argumentos o devolver funciones como resultados. Esto permite una mayor abstracción y reutilización del código.
- **Funciones puras**: Las funciones puras son aquellas que siempre producen el mismo resultado para los mismos argumentos y no tienen efectos secundarios, es decir, no modifican ningún estado externo.
- **Callbacks**: Nombre tradicional con el que se conoce a las funciones que se pasan como argumentos a otras funciones y **se ejecutan en un momento determinado, generalmente cuando ocurre un evento o se completa una tarea**. Por tanto, los callbacks **son un caso concreto de HOF**.
- **Expresiones en lugar de declaraciones**: La programación funcional se basa en la evaluación de expresiones en lugar de la ejecución de declaraciones. Esto significa que el enfoque está en qué se quiere lograr en lugar de cómo lograrlo.

Funcionalidades del lenguaje que vamos a necesitar

Tipos Delegados en CSharp

Ya hemos comentado que vamos a necesitar almacenar una función en una variable para poder guardarla, pasarlas como parámetro a otros métodos o devolverlas como resultado de otros métodos. El **mecanismo variará según el lenguaje** de programación. En C y C++ se usan **punteros a funciones**, en Java o Kotlin se usan **interfaces funcionales (SAM)**, en Python o JavaScript se usan **funciones anónimas (lambdas)** y en C# se usan **delegados**.

Por ello vamos a necesitar saber definir tipos delegados y por tanto **definiremos un delegado**, como **un tipo especial de clase cuyos objetos pueden almacenar referencias a uno o más métodos con la misma signatura** de tal manera que; a través del objeto, sea posible solicitar la ejecución en cadena de todos ellos.

En otras palabras, podemos decir que es un objeto que almacena una o más referencias a un método para ejecutarlo posteriormente.

Sintaxis y ejemplo de uso de delegados

Si un delegado en un objeto, deberá haber un tipo que lo defina. Este tipo tendrá un nombre (identificador) y me indicará la signatura de los métodos que referenciará el objeto delegado.

La sintaxis para definir el tipo será:
`<modificadores> delegate <tipoRetorno> <TipoDelegado>(<parámetros formales>);`

Donde:

- `<TipoDelegado>` será el **nombre del tipo** que me servirá para definir objetos delegado.
- `<tipoRetorno>` y `<parámetros formales>` se corresponderán, respectivamente, con el tipo del valor de retorno y la lista de parámetros formales que definirán la signatura de los métodos cuyas referencias contendrán los objetos de ese tipo delegado.

La sintaxis para instanciar objetos delegado del tipo definido será:
`TipoDelegado oDelegado = IdMetodoQueCumpleLaSignaturaDelTipo;`
que será un *syntactic sugar* del siguiente código...
`TipoDelegado oDelegado = new TipoDelegado(IdMetodoQueCumpleLaSignaturaDelTipo);`

La sintaxis para hacer una llamada al método o métodos que almacena un objeto delegado será:
`tipoRetorno resultado = oDelegado(<parámetros reales>);`
que será un *syntactic sugar* del siguiente código...
`tipoRetorno resultado = oDelegado.Invoke(<parámetros reales>);`
y que realmente estaremos haciendo una llamada al método...
`tipoRetorno resultado = IdMetodoQueCumpleLaSignaturaDelTipo(<parámetros reales>);`

¿No te han quedado claro las diferentes sintaxis?. Veámoslo a través de un ejemplo concreto de uso comentado que puedes descargar de **este enlace**.

Definamos un **tipo delegado** llamado `Operacion` que referenciará a métodos que reciban dos parámetros de tipo `double` y retornen un valor de tipo `double`. Fíjate, sobre todo, que lo que estamos definiendo es un **tipo** como si fuera una clase más.

```
public delegate double Operacion(double op1, double op2);
```

Definamos ahora dos métodos estáticos con cuerpo de expresión que cumplen la signatura definida en el tipo delegado anterior. Esto es, ambos métodos reciben dos parámetros de tipo `double` y retornan un valor de tipo `double`.

```
public static double Suma(double op1, double op2) => op1 + op2;
public static double Multiplica(double op1, double op2) => op1 * op2;
```

Método **`OperaArrays`** que recibe dos arrays de valores y un objeto delegado del tipo **`Operacion`**. El parámetro **`operacion`** indicará la '*estrategia*' (Strategy) a seguir para operar con los valores de ambos arrays. El método devolverá un nuevo array con los resultados de aplicar la operación indicada a cada par de valores de los arrays de entrada.

```
    public static double[] OperaArrays(
                                double[] ops1, double[] ops2,
3                               Operacion operacion)
    {
        double[] resultados = new double[ops1.Length];
        for (int i = 0; i < resultados.Length; ++i)
7           resultados[i] = operacion(ops1[i], ops2[i]);
        return resultados;
    }
```

Programa principal donde se definen dos arrays de valores y se usan ambos métodos definidos anteriormente para realizar sumas y multiplicaciones a través del método **`OperaArrays`**. Fíjate como en la **`línea 6`** estámos guardando las función suma en la variable **`operacion`** del tipo delegado **`Operacion`** y se la pasamos al método **`OperaArrays`**.

En la **`línea 10`** estamos pasando directamente la función multiplicación al método sin necesidad de definir una variable intermedia.

```
    public static void Main()
    {
        double[] ops1 = [ 5, 4, 3, 2, 1 ];
        double[] ops2 = [ 1, 2, 3, 4, 5 ];

6       Operacion operacion = Suma;
7       double[] sumas = OperaArrays(ops1, ops2, operacion);
        Console.WriteLine($"Sumas: {string.Join(" ", sumas)}");

10      double[] multiplicaciones = OperaArrays(ops1, ops2, Multiplica);
        Console.WriteLine($"Multiplicaciones: {string.Join(" ", multiplicaciones)}");
    }
```

🚀 Ampliación Opcional - Multidifusión de delegados

Se producirá cuando un objeto delegado **llama a más de un método** cuando se invoca. Esta cualidad de los delegados nos será útil, más adelante, cuando veamos el concepto de **evento**.

- Para **encadenar** un método / delegado en la multidifusión usaré el `operador +=`
- Para **retirar** un método / delegado de la multidifusión de llamadas usaré el `operador -=`

> ⚠️ **Aviso**
>
> Tiene sentido para **métodos que no retornan nada** (procedimientos), ya que si los delegados retornan algo como en el ejemplo anterior, se asignará el resultado **de la última llamada**.

```
class Ejemplo
{
    // Métodos a añadir al objeto delegado.
    public static void VerSuma(int op1, int op2) =>
    Console.WriteLine($"{op1} + {op2} = {op1 + op2}");
    public static void VerMultiplicacion(int op1, int op2) =>
    Console.WriteLine($"{op1} * {op2} = {op1 * op2}");
    public static void VerDividion(int op1, int op2) =>
    Console.WriteLine($"{op1} / {op2} = {op1 / op2}");

    // Definición del tipo delegado con la signarura de los métodos anteriores.
    public delegate void VerOperacion(int op1, int op2);

    public static void Main()
    {
        // La primera referencia al método a ejecutar la podemos asignar directamente.
        VerOperacion verOperaciones = VerSuma;
        // Las siguientes las añadimos con el operador +=
        verOperaciones += VerMultiplicacion;
        verOperaciones += VerDividion;

        for (int i = 1; i <= 10; ++i)
            // En esta invocación del objeto delegado se realizará una multidifusión
            // a los tres métodos que referencia, ejecutándose los tres.
            verOperaciones(i + 5, i);
    }
}
```

Delegados vs Interfaces en Csharp

De lo visto en este tema, podemos deducir que hay **otra forma de aproximarnos al patrón Strategy** además de usando Interfaces como vimos en temas anteriores.

Vamos a tratar de aproximarnos a ambas a través de un **sencillo ejemplo** de uso ya definido en las BCL. Para ello supongamos la siguiente implementación de la clase `Persona` que hemos usado con anterioridad.

```
public record class Persona(string Nombre, int Edad);
```

Supongamos ahora el siguiente programa principal, donde instanciamos una lista de personas...

```
public class Principal
{
    public static void Main()
    {
        List<Persona> personas =
        [
            new ("Sonia", 35), new ("Antonio", 55), new ("Margarita", 32), new ("Manuel", 50)
        ];
    }
}
```

Si quisiéramos ordenar las personas por `Edad`, la clase list nos va a ofrecer el método `Sort`. Como nuestra clase `Persona` no implementa `IComparable<Persona>`, deberemos indicarle de algún modo al `Sort` la '*estrategia*' de ordenación. Por esta razón `Sort` nos ofrecerá las siguientes sobrecargas ...

1. public void **Sort**(**IComparer<T>?** comparer);
   y si buscáramos la definición del tipo `IComparer<T>` obtendríamos el siguiente **interfaz** parametrizado.

   ```
   public interface IComparer<in T>
   {
       int Compare(T? x, T? y);
   }
   ```

2. public void **Sort**(**Comparison<T>** comparison);
   y si buscáramos la definición del tipo `Comparison<T>` obtendríamos el siguiente **delegado** parametrizado.

   ```
   public delegate int Comparison<in T>(T x, T y);
   ```

Recordemos que si quisiéramos usar el interfaz para ordenar por edad deberíamos definir una clase que implemente el interfaz. Por ejemplo ...

```
public record class Persona(string Nombre, int Edad)
{
    // Definición de la clase que implementa el interfaz
    // con la estrategia de comparación.
    public class ComparaEdad : IComparer<Persona>
    {
        int IComparer<Persona>.Compare(Persona? x, Persona? y) => (x, y) switch
        {
            (null, null) => 0,
            (null, _) => -1,
            (_, null) => 1,
            _ => x.Edad.CompareTo(y.Edad)
        };
    }
}
```

```
public static void Main()
{
    List<Persona> personas =
    [
        new ("Sonia", 35), new ("Antonio", 55),
        new ("Margarita", 32), new ("Manuel", 50)
    ];
    personas.Sort(new Persona.ComparaEdad());
    Console.WriteLine(string.Join(", ", personas));
}
```

Si ejemcutamos el programa principal ahora veremos que las personas se han ordenado por edad. **Mostrando por consola**:

```
Persona { Nombre = Margarita, Edad = 32 }
Persona { Nombre = Sonia, Edad = 35 }
Persona { Nombre = Manuel, Edad = 50 }
Persona { Nombre = Antonio, Edad = 55 }
```

Sin embargo, si quisiéramos usar el delegado `Comparison<T>` tendríamos **una composición de código más sencilla**...

```
// Implementamos un método estático que cumpla la signatura del delegado
// e implemente la estrategia de ordenación.
public static int ComparaEdad(Persona p1, Persona p2) => p1.Edad.CompareTo(p2.Edad);

public static void Main()
{
    List<Persona> personas =
    [
        new ("Sonia", 35), new ("Antonio", 55), new ("Margarita", 32), new ("Manuel", 50)
    ];
    personas.Sort(ComparaEdad);
    Console.WriteLine(string.Join(", ", personas));
}
```

¿Cuándo usar Delegados vs Interfaces en CSharp?

Usaremos delegados cuando:

- Se utilice un modelo de diseño de **eventos**.
- Se prefiere a la hora de encapsular un **método estático o de clase**.
- El autor de las llamadas no tiene ninguna necesidad de obtener acceso a otras propiedades, métodos o interfaces en el objeto que implementa el método.
- Se desea conseguir una **composición sencilla**.
- Una clase puede necesitar más de una implementación del método.

Usaremos interfaces cuando:

- Haya un **grupo de métodos relacionados** a los que se pueda llamar.
- Una clase **sólo necesita una implementación del método**.
- La clase que utiliza la interfaz **deseará convertir esa interfaz en otra interfaz** o tipos de clase.

🎓 **Caso de estudio:**

Supongamos el siguiente programa de ejemplo, donde dado un array de valores. Queremos calcular la media de las raíces cuadradas de los valores y también la media de elevar **e** a los valores.

Una posible propuesta sería la siguiente...

```
class Program
{
    public static double MediaRaiz(double[] valores)
    {
        double total = 0.0;
        for (int i = 0; i < valores.Length; i++) {
            total += Math.Sqrt(valores[i]);
        }
        return total / valores.Length;
    }

    public static double MediaExponencial(double[] valores)
    {
        double total = 0.0;
        for (int i = 0; i < valores.Length; i++) {
            total += Math.Exp(valores[i]);
        }
        return total / valores.Length;
    }

    public static void Main()
    {
        double[] valores = [ 1, 2, 3, 4 ];
        Console.WriteLine("Media raíces:" + MediaRaiz(valores));
        Console.WriteLine("Media exponentes:" + MediaExponencial(valores));
    }
}
```

Sin embargo en la propuesta anterior se repite el código para calcular la media y cómo nos sucedía en otros casos solo se repite la función aplicada al valor.

Piensa cómo sería la solución usando interfaces.
Si no se te ocurre puedes ver la solución el la siguiente página...

```
// Debemos definir el interfaz que implemente la función.
public interface IFuncion
{
    double Funcion(double valor);
}
// Definir tipos que implementen el interfaz con la función específica a aplicar.
public class MediaRaíz : IFuncion
{
    public double Funcion(double valor) => Math.Sqrt(valor);
}
public class MediaExponente : IFuncion
{
    public double Funcion(double valor) => Math.Exp(valor);
}

class Program
{
    // Media ahora recibe el objeto que implementa dicho interfaz.
    public static double Media(double[] puntos, IFuncion funcion)
    {
        double total = 0.0;
        for (int i = 0; i < puntos.Length; i++)
        {
            total += funcion.Funcion(puntos[i]);
        }
        return total / puntos.Length;
    }

    public static void Main()
    {
        double[] puntos = { 1, 2, 3, 4 };
        Console.WriteLine("Media raíces:" + Media(puntos, new MediaRaíz()));
        Console.WriteLine("Media exponentes:" new MediaExponente()));
    }
}
```

Cómo vemos en este caso es más apropiado usar delegados porque tenemos una única función. No hay extensión del interfaz y estamos generando mucho código de definición de tipos a cambio repetir el código de cálculo de la media.

Piensa cómo sería la solución usando delegados.
Si no se te ocurre puedes ver la solución el la siguiente página...

Supongamos el siguiente programa de ejemplo, donde dado un array de valores.

```
class Program
{
    // Definimos el tipo delegado que más adelante incluso nos podremos ahorrar
    public delegate double Funcion(double valor);

    public static double Media(double[] puntos, Funcion funcion)
    {
        double total = 0.0D;
        for (int i = 0; i < puntos.Length; i++) {
            total += funcion(puntos[i]);
        }
        return total / puntos.Length;
    }

    public static void Main() {

        double[] puntos = [ 1, 2, 3, 4 ];
        Console.WriteLine("Media raíces:" + Media(puntos, Math.Sqrt));
        Console.WriteLine("Media exponentes:" + Media(puntos, Math.Exp));
    }
}
```

Delegados Parametrizados (Genéricos)

Podemos definir un delegado de forma parametrizada o que use genéricos. Esto me permitirá usar un **tipos delegado predefinidos para las signaturas de métodos más comunes que se me pueden dar**.

Por ejemplo, es muy común definir delegados en los que se evalúe un parámetro entrada a modo de '*predicado lógico*' que se evaluarán a cierto o falso.

Una posible definición sería...

```
public delegate bool Predicado<T>(T p);
```

y nos permitiría definir un tipo para cualquier signatura de método que evalúe una entrada a modo de predicado. Por ejemplo...

```
// Esté método estático me permite evaluar la si es cierto o no
// el predicado 'Valor es un número par'
public static bool EsPar(int valor) => valor % 2 == 0;

// Esté método estático me permite evaluar la si es cierto o no
// el predicado 'Texto empieza por mayúscula'
public static bool EmpiezaPorMayuscula(string texto) => texto[0] == char.ToUpper(texto[0]);

static void Main()
{
    // En ambos casos puedo usar la misma definición de
    // delegado parametrizado para referenciar a los
    // métodos anteriores.
    Predicado<int> predicado1 = EsPar;
    Predicado<string> predicado2 = EmpiezaPorMayuscula;

    Console.WriteLine(predicado1(4));
    Console.WriteLine(predicado2("Hola"));
}
```

De hecho, ya existe una definición similar en el espacio de nombres `System` de C#...

```
public delegate bool Predicate<in T>(T obj);
```

y es usada en algunos métodos de las BCL **como por ejemplo** el método `public List<T> FindAll(Predicate<T> match);` definido en `List<T>`. De tal manera que podríamos usar nuestro método `EsPar` para obtener todos los números pares en una lista de números de la siguiente manera ...

```
List<int> l = [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ];
Console.WriteLine(string.Join(", ", l.FindAll(EsPar)));
```

Además de `delegate bool Predicate<in T>(T obj)` podemos destacar la siguientes definiciones en las BCL de delegados parametrizados...

Definiciones para procedimientos `Action`

Bajo el tipo `Action` tendremos predefinidos delegados que admitirán métodos que no retornan nada (void) y que podrán tener de 0 a 16 parámetros.

```
// Si no parametrizamos es porque no habrá parámetros de entrada.
public delegate void Action()
...
public delegate void Action<in T1, in T2, …, in T16>(T1 obj1, T2 obj2, …, T16 obj16)
```

Un ejemplo de uso de este delegado sería el método `public void ForEach(Action<T> action);` definido en `List<T>`. De tal manera que me permitirá recorrer los elementos de la secuencia e ir aplicando a cada uno de ellos la acción especificada en forma de delegado...

```
public static void Muestra(int valor) => Console.WriteLine($"{valor:D2}");

static void Main()
{
    List<int> valores = [ 2, 6, 3, 8, 2 ];

    // Defino un delegado para métodos con un solo parámetro de entrada entero.
    Action<int> accion = Muestra;
    valores.ForEach(accion);
    // o directamente ... valores.ForEach(Muestra);
}
```

Definiciones para funciones `Func`

Bajo el tipo `Func` tendremos predefinidos delegados que admitirán métodos que no retornan un tipo a modo de '*función*' y que como los `Action` podrán tener de 0 a 16 parámetros.

```
// Fíjate que ahora la parametrizacion define al menos el tipo R de (Retorno) de la función.
public delegate R Func<out R>()
...
// El tipo de retorno se definirá siempre al final en la parametrización.
public delegate R Func<in T1, …, in T16, out R>(T1 obj1,  …, T16 obj16)
```

Iremos encontrando este tipo de delegado más adelante, en ciertas definiciones de las BCL de `System.Linq`. Para poner nuestro ejemplo, vamos a fijarnos que el tipo delegado `Predicado<T>` que definimos en el nuestro primer ejemplo. Se puede definir también con `Func` de la siguiente manera `Func<T, bool>`. Aunque ambos delegados admitan métodos con la misma signatura, para C# serán tipos diferentes.

Veamos cómo reescribir el ejemplo de los predicados usando `Func`...

```
public static bool EsPar(int valor) => valor % 2 == 0;
public static bool EmpiezaPorMayuscula(string texto) => texto[0] == char.ToUpper(texto[0]);

static void Main()
{
    // En ambos casos al ser el último tipo parametrizado un bool
    // estaremos indicando que la signatura de los métodos debe retornar eso.
    Func<int, bool> predicado1 = EsPar;
    Func<string, bool> predicado2 = EmpiezaPorMayuscula;

    Console.WriteLine(predicado1(4));
    Console.WriteLine(predicado2("Hola"));
}
```

Aunque solo se dará en casos aislados, a lo mejor queremos definir un delegado que referencie a un método con más de un parámetro de salida. En este caso, si nos acordamos de principio de curso, usaremos tuplas para definir el valor de retornos.

Recordemos el ejemplo que vimos para este caso...

```
static (double sen, double cos) Direccion(double anguloGr)
{
    double anguloRad = anguloGr * Math.PI / 180;
    return (Math.Sin(anguloRad), Math.Cos(anguloRad));
}
```

No obstante, como comentamos, a partir de C#7 no sería necesario. Si quisiéramos usar `Func` para definir un tipo delegado que referenciase a los métodos anteriores podríamos hacer lo siguiente...

```
static void Main()
{
    Func<double, (double, double)> f = Direccion;
    (double seno, double coseno) = f(90);
    Console.WriteLine($"s={seno:F2}, c={coseno:F2}");
}
```

Tipos Anónimos Inmutables en CSharp

Es una **herramienta muy útil en programación funcional** para **crear objetos ligeros y temporales** que no requieren una definición de clase completa. Son especialmente útiles en escenarios donde se necesita agrupar datos de manera rápida y sencilla, como consultas sobre una secuencia de datos o en operaciones de **transformación de datos**. A esta **agrupación o proyección de datos** resultado de una consulta sobre otros objetos se le denomina **DTO (Data Transfer Object)** y deben tener las **características de inmutabilidad y comparación de igualdad que tienen los tipos anónimos** que teníamos en C# a través de `record class`. Pero en ocasiones, no queremos definir un tipo completo para ello porque los necesitamos en un alcance (scope) local o para un proceso de mapeo intermedio. Es aquí donde los tipos anónimos nos serán de gran utilidad.

Resumiendo, **definiremos los tipos anónimos como** una manera de encapsular un conjunto de propiedades de solo lectura en un único objeto sin tener que definir un tipo. Para ello, **el compilador genera el nombre del tipo de forma transparente para el programador y por tanto no disponible en el nivel de código fuente**.

En el **ejemplo** siguiente se muestra dos tipos anónimo que se inicializa con una propiedad `Name` el primero y dos propiedades `Nombre` y `Edad` el segundo.

```
var estudianteDesconocido  = new { Name = "Rigoberto" };
var estudianteDesconocido2 = new { Nombre = "Pedro", Edad = "12" };
```

Podemos ahorrarnos el indicar los nombres de las propiedades si utilizamos **identificadores de variables** para inicializar el tipo anónimo ...

```
double X = 9.1;
float Y = 3.2;

// La variable point1 tendrá una propiedad llamada X de tipo double y
// otra llamada Y del tipo float.
var point1 = new { X, Y };

// Incluso podemos combinar las formas de inicializar,
// en este el siguiente caso point2 tiene como propiedades X y SuperY.
var point2 = new { X, SuperY = Y };
```

También se puede definir un array de elementos con tipo anónimo, combinando una variable local con tipo implícito y una matriz con tipo implícito. Por ejemplo ...

```
var fruitsSize = new[]
{
    new { Name = "Apple", Diameter = 4 },
    new { Name = "Grape", Diameter = 1 }
};
```

> **Importante**
>
> No podremos utilizar expresiones de coleccion var `coleccion = [...]`
> Además, en este caso hemos creado **dos objetos anónimos con el mismo nombre de propiedades** y por tanto del '*mismo tipo*'. Esto será detectado internamente por el Runtime que **combinará ambos tipos anónimos creados**, en uno solo.

Restricciones de los tipos anónimos:

- Son inmutables.
- No pueden definir Propiedades Privadas
- No pueden definir métodos, campos, constructores, etc.
- No se pueden inicializar a null.
- Al carecer de un nombre de tipo, no se pueden retornar desde un método o pasar como parámetros a los mismos.

Al usar `Equals(object obj)` **se considerará iguales** aquellos métodos anónimos que tengan:

- Las mismas propiedades, en nombre y número.
- El mismo orden de declaración de las propiedades.
- Los mismos valores para esas propiedades.

```
var anonimo1 = new { Nombre = "María", Edad = 23 };
var anonimo2 = new { Nombre = "María", Edad = 23 };

// Mostrará True a pesar de que sean referencias a
// objetos diferentes en memoria.
Console.WriteLine(anonimo1.Equals(anonimo2));
```

Por último, puesto que heredan de la clase `object` podrán mostrarse `string ToString()` por ejemplo...

```
var anonimo = new { Nombre = "María", Edad = 23 };

// Mostrará "{ Nombre = "María", Edad = 23 }" sin necesidad de invalidar ToString().
Console.WriteLine(anonimo);
```

Cálculo Lambda

Es uno de los pilares de la programación funcional. Esté fue formulado por el matemático-lógico norteamericano **Alonzo Church** y consiste en '*...un sistema formal diseñado para investigar la definición de función, la noción de aplicación de funciones y la recursión...*'.

Se basa en una definición de función alternativa a la tradicional que dentro del contexto de las ciencias de computación denominaremos: **'*Expresión Lambda*'**.

Definición de Expresión Lambda

Para entender lo que es, vamos ver el concepto de función dependiendo de ámbito de conocimiento...

1. En **matemática tradicional** las funciones se representan con un nombre f o g

Definición	**Explicación**
$f(x) = x + 2$	Un parámetro.
$pow(x, y) = x^y$	Dos parámetros.

2. En la **teoría matemática del cálculo lambda** (*λ-calculus*) esta misma función **se expresaría sin nombre**, solo los parámetros que entran y la expresión que la representa.

Definición	**Explicación**
$\lambda x.x + 2$	Un parámetro (definición) y equivale a $f(x) = x + 2$.
$(\lambda x.x + 2)10 = 12$	Un parámetro (aplicación de una valor) y equivale a $f(10) = 10 + 2 = 12$.
$\lambda(x, y).x^y$	Dos parámetros.

3. ✋ En **ciencias de computación** se expresará a través de una **función anónima o expresión lambda**.

Definición	**Explicación**
$x \rightarrow x + 2$	Un parámetro (definición) y equivale a $f(x) = x + 2$.

Definición	Explicación
$(x \to x + 2)(10) = 12$	Un parámetro (aplicación de un valor) y equivale a $f(10) = 10 + 2 = 12$.
$(x, y) \to x^y$	Dos parámetros.

Esta última, forma de definir funciones de forma anónima y como expresiones, es la que usaremos el la gran mayoría de lenguajes de programación y aunque vamos a ver el caso concreto de C#, la forma de hacelos será muy similar en otros lenguajes. Por
ejemplo, si ya las hemos usado en JavaScript, su sintaxis nos será muy familiar.

Expresiones Lambda en CSharp

Una **λ-expresión** en C# es un **método anónimo** (sin nombre) que se ha de declarar/definir y asignar inmediatamente, sobre una instancia de un **delegado**.

Las λ-expresión puede ser usadas en alguno estos escenarios en C#:

1. Definir cuerpos de expresión dentro de una clase (**Esto ya lo hemos usado**). Por ejemplo, **un cuerpo de expresión es una expresión lambda**.
2. Pasar como argumento a otra función.
3. Como tipo de retorno de una función, esto es, '*Una función que retorna otra función*.
4. Asignación a instancias de delegados parametrizados.

La representación general de una expresión lambda sigue el siguiente formato general...

`(parámetros) => sentencia o {bloque de sentencias con return si función}`

Vamos a describir que es cada cosa:

1. `(parámetros)`: Lista de parámetros separada por comas.

   ```
   (x, y) => Math.Pow(x, y);
   ```

   Se debe dejar los paréntesis vacíos `()` si no hay parámetros.

   ```
   () => Console.WriteLine("Hola");
   ```

   Si solo hay **un parámetro**, los paréntesis se pueden obviar.

```
// Suponiendo que s es de tipo string la expresión lambda siguiente
// se evaluará a tipo delegado Func<string, int> esto es, una función anónima
// que recibe un string y retorna un int.
s => s.Lenght;
```

2. `=>`: Este es el **operador** que se encarga de mapear los parámetros a la expresión o al conjunto de sentencias del bloque.
3. La parte derecha, tras el operador `=>`, puede ser una única sentencia en forma de expresión que se evalúe a lo que retorna la función como en lo ejemplo anteriores o de forma menos habitual, un bloque de instrucciones. En este último caso la sintáxis será equivalente a la de las funciones tradicionales y tratando de sangrar el código y escribirla en varias líneas para mayor claridad.

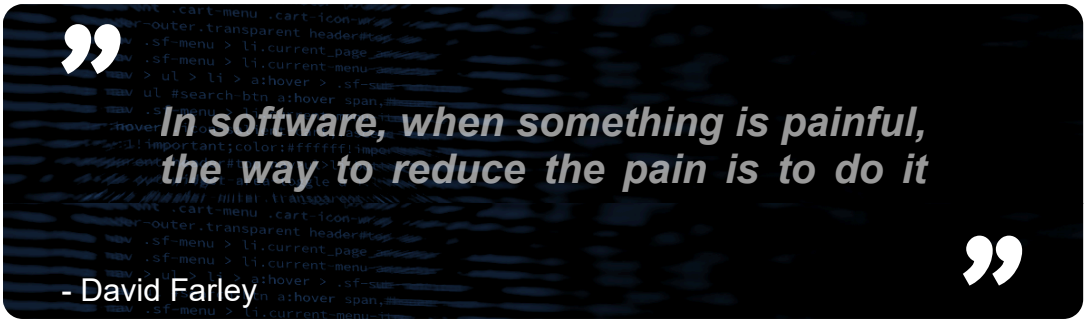
```
    (x, y) =>
    {
        Console.WriteLine($"Calculando {x} elevado a {y})");
4       return Math.Pow(x, y); // Fíjate que ahora sí debemos usar return.
    }
```

Las expresiones lambda anteriores anteriores de forma análoga a lo que sucedería con la expresión `int x = 3 + 4`, deberemos asignarla a algo. Puesto que los tipos que referencian a funciones son los objetos delegado, deberemos asignarlo a algún identificador de objeto delegado parametrizado o no. Es más, **será completamente necesario para deducir los tipos de los parámetros de entrada y de retorno**.

```
// x es double y retorna double
Func<double, double> cubo = (x) => x * x * x;

// x , y son enteros y retorna entero
Func<int, int, int> suma = (o1, o2) => o1 + o2;

// ERROR no sé deducir los tipos de forma implícita.
// Esto solo sería válido en un lenguaje débilmente tipado como JavaScript.
var suma = (o1, o2) => o1 + o2;
```



Clausuras en CSharp

Para entender el concepto de **clausura**, vamos a ver los siguiente definición del lenguaje C#.

Variables Externas Capturadas

En el cuerpo de expresión de una expresión lambda, podemos incluir referencias a variables locales y a los parámetros de un método. A estas referencias se les conoce como **variables externas capturadas**.

```
static void A()
{
    // Variable local factor, su alcance o scope es la el método A
    // fuera del mismo la variable dejará de existir.
    int factor = 3;

    // factor es una variable externa capturada en la expresión lambda pues
    // está siendo usada en un ámbito más cercano.
    Func<int, int> producto = n => n * factor;

    // Como hago una invocación del delegado dentro del scope de A
    // factor sigue existiendo y valiendo 3 por lo que al ejecutarse
    // el cuerpo de expresión de la función lambda a la que referencia
    // se mostrará 5 * 3 = 15
    Console.WriteLine(producto(5));
}
```

A las **expresiones lambda** que están integradas por variables externas capturadas se les llama **clausuras**.

Este tipo de variables, se evalúan en el momento en que se llama a la expresión Lambda de tal manera que si reescribiéramos el código anterior así ...

```
    static void A()
    {
        int factor = 3;
        Func<int, int> producto = n => n * factor;
5       factor = 10;

        // Ahora mostrará 5 * 10 = 50 porque evalúa el valor de factor en
        // el momento de la llama a la función anónima.
        Console.WriteLine(producto(5)); // Mostrará 50
    }
```

La referencia y el valor a una variable capturada, **está disponible mientras el ámbito de la expresión lambda** esté al alcance de su ejecución. Esto es, a pesar de encontrarnos fuera el ámbito de definición de la variable externa capturada, se guarda una referencia y siempre estará accesible para la expresión lambda.

Pero, en nuestro ejemplo `producto` que es la referencia a nuestra expresión lambda y `factor` que es la variable externa capturada, tienen el mismo ámbito de ejecución que es el método `A`, esto es, ambas referencias existen en el heap mientras estoy ejecutando `A` y al finalizar el método desaparecen. Entonces... **¿Cómo puede extenderse el ámbito de ejecución de `producto` más allá del ámbito de existencia de `factor`?**

La respuesta es, mediante las ya mencionadas **funciones de orden superior** o **HOF** (High-Order Functions). Recuerda que las HOF en el λ-cálculo decíamos que **son funciones que reciben como parámetro o retornan otra función**.

¿Cómo pasamos por parámetro o devolvemos una función en C#?. La respuesta sería a través de un tipo delegado. Entonces, cuando tratamos los delegados ... ¿Aquellos métodos que recibían un delegado eran HOF?. La respuesta es **sí**.

¿Cómo redactaremos el código anterior para que el método `A` sea una HOF y la función lambda (clausura) tenga un alcance de ejecución más allá de su ámbito de definición?. Una posible propuesta sería la siguiente ...

```
// Ahora el método A es HOF porque retorna una función
static Func<int, int> A()
{
    int factor = 3;
    return n => n * factor;
}
```

```
static void Main()
{
    var producto = A();
    // A() ha desvuelto el control a Main y se supone que la variable local factor
    // ya ha desaparecido. Sin embargo, la clausura que la capturaba aún está
    // referenciada por el delegado producto y por tanto aún la podemos ejecutar.
    // Al no desaparecer el alcance de ejecución la variable capturada factor aún
    // no ha desaparecido de la memoria y tendrá el último valor que se le asignó
    // que es 3.
    Console.WriteLine(producto(5)); // Mostrará 15
}
```

Resumen

Es momento de realizar una breve recapitulación de conceptos...

En programación funcional (λ-cálculo) se denominan funciones de orden superior (HOF), aquellas que:

1. Reciben como parámetro otra función $(r_1 \to r_2) \to r_3$
2. Devuelven como parámetro otra función $r_1 \to (r_2 \to r_3)$
3. O ambas cosas $(r_1 \to r_2) \to (r_3 \to r_4)$.

En C# lo serán a aquellas funciones que reciban o devuelvan un delegado, o ambas cosas.

1. Reciben un delegado...

```
Func<int, int> func1 = d => d + 1;
Func<Func<int, int>, int, int> func2 = (func, d) => func(d) + 2;
// func2 es una HOF
int a = func2(func1, 3); // a = func1(3) + 2 = (3 + 1) + 2 = 6
```

2. Devuelven un delegado...

```
Func<int, Func<int, int>> suma = op1 => op2 => op1 + op2;
int a =  suma(3)(2); // a = (op2 => 3 + op2)(2) = 3 + 2 = 5
```

Veamos un **par de ejemplos** de uso de este concepto de clausura con una utilidad más concreta pero simplemente para que terminemos de entender el concepto y la terminología ...

Ejemplo 1:

Definiremos una HOF `Contador()` que me devuelva una clausura que al llamarla me devuelva un contador capturado incrementado.

```
public static Func<int> Contador()
{
    int i = 0;
    return () => ++i; // Devuelvo la clausura que captura el contador i..
}

public static void Main()
{
    // Cada cuenta captura una variable i local diferente por lo que
    // llevarán cuentas independientes
    Func<int>  cuenta1 = Contador(); // Creo un contador
    Func<int>  cuenta2 = Contador(); // Creo un segundo contador contador

    Console.WriteLine(cuenta1()); // Muestra 1
    Console.WriteLine(cuenta2()); // Muestra 1 por ser otro contador
    Console.WriteLine(cuenta2()); // Muestra 2
    Console.WriteLine(cuenta2()); // Muestra 3
    Console.WriteLine(cuenta1()); // Muestra 2
}
```

Ejemplo 2:

Definiremos una HOF `Potencia(double y)` que me permita definir otras funciones que calculen potencias con un

```
// HOF que devuelve una clausura que captura el parámetro de entrada 'y'.
// Fíjate que estemos definiendo un cuerpo de expresión que se evalúa una
// clausura que dada una base 'x' calcula 'x' elevado a la variable capturada 'y'.
public static Func<double, double> Potencia(double y) => x => Math.Pow(x, y);

// Definimos una función que calcula cuadrados pasándole el exponente a capturar.
public static readonly Func<double, double> Cuadrado = Potencia(2);
// Definición análoga para capturar el cubo.
public static readonly Func<double, double> Cubo = Potencia(3);

public static void Main()
{
    Console.WriteLine(Cuadrado(10)); // x = 10 elevado a y = 2 mostrará 100
    Console.WriteLine(Cubo(10));     // x = 10 elevado a y = 3 mostrará 1000
}
```

🚀 **Ampliación Opcional:** ¿Cómo se implementaría el código anterior en **JavaScript** usando **funciones-λ** y **clausuras**?

```
const potencia = (y) => (x) => Math.pow(x, y);

const cuadrado = potencia(2);
const cubo = potencia(3);

console.log(cuadrado(10));
console.log(cubo(10));
```

Como vemos **el código es muy similar** pero al **no tener tipos** no necesitamos declarar los tipos de los parámetros ni el tipo de retorno.

Patrón Map-Filter-Fold

Si vamos al principio del tema donde definíamos lo que era la programación funcional, decíamos que:

> '*La **programación funcional** es un **paradigma de programación declarativa** basado*' ...

Para entender un poco mejor a que se refiere con '***programación declarativa***', vamos a comparar y resumir de forma sencilla la diferencia entre la programación imperativa y declarativa. Diremos que, en la **imperativa** le decimos nosotros al ordenador **'*cómo tiene que hacer algo*'** mediante una secuencia de instrucciones que sería más o menos lo que hemos hecho hasta ahora. Sin embargo, en la **declarativa** le decimos al ordenador **'*lo que tiene que hacer*'**, proporcionándole únicamente las estrategias para hacerlo.

Por poner un ejemplo, **el lenguaje SQL sería declarativo** ya que en él, le decimos al SGBD por ejemplo que seleccione n-tuplas que cumplan un criterio y les aplique una estrategia de agrupación, pero **'*no le indicamos cómo tiene que hacerlo*'**, de hecho no sabemos cómo lo hace internamente.

Sin entrar en aspectos más teóricos o formales matemáticos como quizá hemos hecho al hablar del λ-cálculo, diremos que **la gran mayoría de lenguajes modernos proporcionan una serie de métodos para trabajar sobre secuencias de datos de forma similar a cómo lo haríamos en SQL aplicando programación funcional**.

En C# estos métodos para trabajar con secuencias (`IEnumerable<T>`) están definidos como métodos de extensión en `System.Linq` como ya comentamos al hablar de este tipo de métodos.

Entre ellos podremos destacar: `Select` , `Where` , `Agreggate` , `GroupBy` , `Distinct` , `ElementAt` , `Join` , `GroupJoin` , `OrderBy` , `Reverse` , `SelectMany` , etc. **¿Te suenan?**

Map-Filter-Fold (o también conocido como **Map-Reduce**) es un patrón muy común en programación funcional para **transformar, filtrar y agregar datos en secuencias** que utiliza alguna de las funciones de extensión definidas para trabajar con secuencias en C#. En concreto, **Select-Where-Aggregate**.

Vamos a encontrarlo en numerosos lenguajes tan populares como **JavaScriprt**, **Java**, **Python**, **Kotlin**, **Rust**, etc y que permite hacer operaciones declarativas sobre secuencias de elementos de forma análoga a SQL.

> **Nota**
>
> Detrás de este patrón hay una gran teoría matemática aunque nosotros la vamos a obviar y vamos a tratar de explicar las operaciones de forma más didáctica. Si quieres profundizar en el tema puedes seguir los enlaces en el tema.

> Además, durante el resto del tema **vamos a utilizar la siguiente notación** para representar secuencias (colecciones) y objetos...
>
> - **Corchetes** para secuencias **[** e1, e2, e3, ..., en **]** o **Sec[** e1, e2 **]**
> - **Llaves** para objetos **{** Propiedad1=valor1, Porpiedad2=valor2 **}**

Map (o Select)

Map es una HOF que aplica una función única de '*mapeo*' o **transformación** a cada elemento de la secuencia y devuelve una nueva secuencia que contiene los resultados de la transformación en el mismo orden de la secuencia de entrada.

A lo largo de este tema vamos a ejemplificar los conceptos de forma genérica con emojis, porque realmente es un concepto abstracto y estos nos ayudarán a entender la abstracción de forma didáctica.
Supongamos que representamos **map** con la expresión **map = (A → B) × Sec[A] → Sec[B]** donde:

- **A** son productos en crudo o sin procesar.
  Ej. el maíz 🌽.
- **B** son productos cocinados o procesados.
  Ej. las palomitas de maíz 🍿.
- **A → B** sería la función que cocina o **transforma** un producto en crudo en un producto cocinado y la podemos llamar por ejemplo **cook**. Esta función se pasará como parámetro a la función **map** (por eso **map** es una HOF).
  Ej. **cook(** 🌽 **)** → 🍿.
- **Sec[A]** seria la secuencia de **entrada** de la función **map** de productos en crudo.
  Ej. [🐮, 🐟, 🥓, 🐔, 🌽]
- **Sec[B]** seria la secuencia de **salida** de la función **map** de productos ya cocinados o procesados por la función **cook**.
  Ej. [🍔, 🍥, 🍟, 🍗, 🍿]

Con lo cual podríamos decir que, **map = cook × ProductosCrudos → ProductosCocinados**. Esto es, map dada una secuencia de productos en crudo le aplica a cada producto la **función de transformación** cocinar y me dará como resultado una secuencia de productos cocinados.
Ej. **[**🐮, 🐟, 🥓, 🐔, 🌽**].map(cook) → [**🍔, 🍥, 🍟, 🍗, 🍿**]**

En C# lo aplicaremos a través del método extensor **Select** ...

```
// Se aplica a secuencia de entrada IEnumerable<A>
// A cada elemento de la secuencia de entrada se le aplica el 'mapeo' A => B
// Retorna una secuencia de salida IEnumerable<B> con los
// elementos de la secuencia de entrada mapeados.
IEnumerable<B> Select<A, B>(this IEnumerable<A> source, Func<A, B> mapeoDeAenB);
```

Veamos su uso a través de un ejemplo sencillo donde transformamos una secuencia de números reales en su equivalente entero. '*Mapeando*' de real a entero cada uno de elementos de la secuencia de entrada.

```
// Dada la siguiente secuencia de números reales
double[] reales = [ 1.3, 3.4, 4.5, 5.6, 8.7 ];

// Aplicamos la función ToInt32 que cumple con el tipo delegado Func<double, int>
// para transformar la secuencia a enteros.
IEnumerable<int> enteros = reales.Select(Convert.ToInt32);

Console.Write(string.Join(", ", enteros));
```

Filter (o Where)

Filter es una HOF que aplica un predicado a cada elemento de la secuencia de entrada y devuelve una secuencia de salida con los elementos en la secuencia que cumplen el predicado.

Supongamos que representamos **filter** con la expresión **filter = (A → bool) × Sec[A] → Sec[A']** donde:

- **A** son productos cocinados o preparados.
  Ej. una hamburguesa 🍔.
- **A → bool** sería el predicado que evalúa si los elementos de **A** cumplen un determinado criterio o no. Este predicado se pasará como parámetro a la función **filter**.
  Ej. **isVegetarian(🍔) → false**.
- **Sec[A]** seria la secuencia de **entrada** de la función **filter** de productos ya cocinados.
  Ej. [🍔, 🍥, 🍟, 🍗, 🍿]
- **Sec[A']** seria la secuencia de **salida** de la función **filter** de productos ya cocinados que cumplen el predicado **isVegetarian**, esto es, $A' \subseteq A$.
  Ej. [🍟, 🍿]

Con lo cual podríamos decir que, **filter = isVegetarian × ProductosCocinados → ProductosCocinadosVegetarianos**. Esto es, filter dada una secuencia de productos ya cocinados me devolverá aquellos que sean vegetarianos.
Ej. **[🍔, 🍥, 🍟, 🍗, 🍿].filter(isVegetarian) → [🍟, 🍿]**

En C# lo aplicaremos a través del método extensor **Where** ...

```
// Dada la siguiente secuencia de enteros.
int[] enteros = [ 1, 3, 4, 5, 8 ];

// Filtramos aquellos que sean pares...
// 1. Definimos el predicado para ver si un número es par.
Func<int, bool> esPar = n => n % 2 == 0;
// 2. Aplicamos el predicado a la función where
IEnumerable<int> enterosPares = enteros.Where(esPar).ToList();
Console.Write(string.Join(", ", enterosPares));
```

Fold (o Aggregate)

Fold es una HOF también conocida como **reduce**, **aggregate**, **accumulate**, etc. que '*combina*' los elementos de la secuencia de entrada reduciéndolos a un único elemento de retorno. Para obtener este elemento de retorno necesitaremos una función binaria que combine los elementos de la secuencia en dicho elemento de retorno que normalmente tendrá un estado inicial.

Supongamos que representamos **fold** con la expresión **fold = (B × A → B) × Sec[A] × B → B** donde:

- **B** es el elemento de retorno el cual tomará un valor inicial.
  Ej. Pepe que **inicialmente** está hambriento y cansado 😫 pero puede tener otros estados como satisfecho 😋
- **A** son los productos cocinados o preparados.
  Ej. la hamburguesa de vacuno 🍔
- **Sec[A]** obviamente es una secuéncia de productos cocinados o preparados.
  Ej. [🍔, 🍥, 🍟, 🍗, 🍿]
- **B × A → B** función binaria que combina **B** con **A** y se evalúa a **B**.
  Ej. En nuestro ejemplo, esta función de combinación puede ser **eat(😫, 🍔) → 😋** donde el resultado de combinar a 😫(Pepe hambriento) con la 🍔 (hamburguesa) sería 😋(Pepe satisfecho). Ojo!, que también puede pasar que al comer (combinar), el estado de pepe no cambie y siga hambriento **eat(😫, 🍔) → 😫**
  En otras palabras, **eat** es una función de plegado (**fold**), reducción (**reduce**), agregación (**aggregate**), etc. de la hamburguesa sobre Pepe y que me da como resultado al propio Pepe con menos hambre o saciado...

Con lo cual podríamos decir que, **fold = come × ProductosCocinados × PepeHambriento → PepeSatisfecho**. Esto es, fold dado un Pepe hambriento y una secuencia de productos cocinados, me devolverá un Pepe satisfecho.
Ej. **[🍟, 🍿].fold(😫, eat) → 😋**

En C# lo aplicaremos a través de las sobrecargas del método extensor **Aggregate** ...

> **Nota**
>
> `System.Linq` nos proporciona muchas funciones de agregación ya implementadas como métodos de extension sobre secuencias tales como: **Sum**, **Max**, **Min**, **Average**, o **Count**. Quen realidad son casos particulares de `Aggregate` que se nos darán un 90% de las ocasiones. Evitando así la complejidad de tener que definir la función de agregación.

```
// Dada la siguiente secuencia de notas enteras
int[] notas = [ 10, 3, 4, 5, 8, 2 ];

// Cuenta las notas mayores o iguales a 5.
// 1. Definimos la función binaria de agregación que vamos apasar a Aggregate....
//    c -> Es el contador.
//    n -> Es un elemento de la secuencia (una nota).
//    Se evalúa a la cuenta incremetada o sin incrementar dependiendo de
//    si nota >= 5 o no.
Func<int, int, int> cuentaAprobados = (c, n) => n >= 5 ? c + 1 : c;

// 2. Aplicamos la HOF de agregación, pasándole el valor inicial de la cuenta y
//    la función agregadora.
int aprobados = notas.Aggregate(0, cuentaAprobados);
Console.WriteLine($"Total aprobados: {aprobados}");
```

`Aggregate` por defecto recorrerá la secuencia de izquierda a derecha y para el caso anterior de contar aprobados nos daría igual. Sin embargo, muchos lenguajes funcionales ofrecen ambas posibilidades:

- `foldl` : (fold left) Recorre como `Aggregate` de izquierda a derecha.
- `foldr` : (fold right) Recorre la secuencia de derecha a izquierda.

Veamos una ejemplo de como hacerlo en C#. Para ello, vamos a suponer que queremos implementar el método de utilidad definido en la clase `String` y que hemos usado muchas veces a lo largo del curso. `string.Join(string? separador, IEnumerable<string?> textos)`.

Si nos fijamos cumple con **(B × A → B) × Sec[A] × B → B** pero ahora no obtendremos el mismo resultado si hacemos `foldl` o `foldr` ...

- **A** Serán una palabra (`String`) a concatenar.
- **Sec[A]** Será la secuencia de palabras a concatenar.
- **B** Ahora el tipo *'agregador'* sería un `StringBuilder` donde vamos a concatenar todas las palabras con el separador adecuado.
    i. Valor inicial será `new StringBuilder(palabras.Length > 0 ? palabras.First() : "")`, esto es, un `StringBuilder` vacío si la secuencia esta vacía o uno que contenga la primera palabra de

la misma si contiene alguna.
   ii. Valor final será el `StringBuilder` inicial `B` al que le he concatenado todas las palabras por la derecha.
- **(B × A → B)** La función binaria de agregación será de tipo `Func<StringBuilder, string, StringBuilder>` y recibirá el `StringBuilder` `B` con lo que llevo de la cadena concatenada, la siguiente palabra **A** a concatenar con el separador y retornará el `StringBuilder` **B** con la palabra concatenada por la derecha.

Podemos resumir de forma simple que `Join` se comporta como `foldl` porque entra una secuencia de cadenas y devuelve un único tipo resultado de concatenar las cadenas con un separador determinado.

La única diferencia es que `Join` devuelve un `string` y nosotros vamos a devolver un `StringBuilder` por eficiencia.

Veamos pues nuestra propuesta de implementación de `Join` con un `Aggregate` (`foldr`) en C# ...

```
// Definimos la secuencia
string[] palabras = { "Once", "upon", "a", "time" };

// Definimos el separador que será una variable capturada en la función lambda.
const string separador = " ";

// Función lambda agregadora concatena...
//  Si la palabra a concatenar no es nula y no coincide con el valor inicial de la
//  concatenación o en otras palabras es la primera palabra de la secuencia la
//  concatenamos con el valor del separador capturado.
Func<StringBuilder, string, StringBuilder> concatena =
(f, p) => f.Append(p != null && f.ToString() != p ? $"{separador}{p}" : "");

StringBuilder frase = palabras.Aggregate(
                        new StringBuilder(palabras.Length > 0 ? palabras.First() : ""),
                        concatena);

Console.WriteLine(frase); // Mostrará "Once upon a time"
```

pero...¿y si quisiéramos hacer un `foldr` en C#?. Sería '*tan simple*' como **invertir la secuencia de entrada** con `Reverse()` y el lugar de tomar la primera ocurrencia (`First`) de la secuencia para inicializar nuestro `StringBuilder` , tomaríamos la última `Last` .

```
StringBuilder frase = palabras
                    .Reverse()
                    .Aggregate(
                        new StringBuilder(palabras.Length > 0 ? palabras.Last() : ""),
                        concatena);

Console.WriteLine(frase); // Mostrará "time a upon Once"
```

`Aggregate` además tiene varias sobrecargas y vamos a comentar una de ellas que nos puede ser interesante en este caso.

Si vamos a ver la definición de `Aggregate` para el uso anterior tendremos el siguiente interfaz que ya deberíamos entender...

```
// TSource es la parametrización de A
// TAccumulate es la parametrización de B
public static TResult Aggregate<TSource, TAccumulate>(
        this IEnumerable<TSource> source,              // Sec[A]
        TAccumulate seed,                              // B (inicial)
        Func<TAccumulate, TSource, TAccumulate> func);  // B × A → B
```

Sin embargo, tenemos también la sobrecarga ...

```
public static TResult Aggregate<TSource, TAccumulate, TResult>(
        this IEnumerable<TSource> source,
        TAccumulate seed,
        Func<TAccumulate, TSource, TAccumulate> func,
        Func<TAccumulate, TResult> resultSelector);
```

donde aparece otra nuevo tipo parametrizado `TResult` y función de entrada `Func<TAccumulate, TResult> resultSelector` que una vez realizado todo el proceso de **'*fold*'** me transformará (mapeará) mi `TAccumulate` final en un `TResult`.

En nuestro ejemplo, al definir el `TAccumulate` como un `StringBuilder` por optimización, el resultado del `Aggregate` será un `StringBuilder` y no un `string` como nos proporcionaría el `string.Join(...)`. Pero la función de mapeo `Func<TAccumulate, TResult> resultSelector` nos permitirá convertir el `StringBuilder` a un `string` con `sb => sb.ToString()` y será lo que terminará devolviendo `Aggregate`.

Nuestra expresión equivalente al `Join` haciendo un `Aggregate` (`foldl`) finalmente quedaría ...

```
string[] palabras = { "Once", "upon", "a", "time" };
const string separador = " ";

// Con Join
string frase = string.Join(separador, palabras);

// Con Aggregate
string frase = palabras.Aggregate(
    new StringBuilder(palabras.Length > 0 ? palabras.First() : ""),
    (f, p) => f.Append(f.ToString() != p && p != null ? $"{separador}{p}" : ""),
    sb => sb.ToString());

Console.WriteLine(frase);
```

Fíjate que en esta ocasión hemos pasado las funciones de agregación y mapeo directamente a `Aggregate` en lugar de asignarlas previamente a un delegado. Esto es posible porque **C# es capaz de deducir los tipos** de `(f, p) =>` y de `sb =>` a partir del tipo de la secuencia `palabras` y el tipo del objeto que le pasamos como acumulador.

> **Nota**
>
> Obviamente en este caso, es muchísimo más simple utilizar la función ya implementada `Join`. Pero hemos puesto este ejemplo, además de para ver el concepto de `foldl` y `foldr`, para entender la potencia de esta función si entendemos bien lo potente que es esta operación conceptualmente.
> De momento la hemos usado para cosas tan diferentes como: que Pepe coma 😋, contar aprobados o concatenar palabras con un separador.

Combinando las tres operaciones

Podemos combinar dichas operaciones para realizar nuestro proceso en una única función de forma declarativa.

Imaginemos nuestro ejemplo en que queremos alimentar a Pepe, que recordemos que es vegetariano y partimos de un montón de productor en crudo. Podríamos hacer...

[🐮, 🐟, 🍆, 🐔, 🌽].map(cook).filter(isVegetarian).fold(🤮, eat) → 😋

Esto equivaldría a las operaciones anteriores de forma encadenada. Donde la secuencia de salida de una función pasa a ser la de entrada de la siguiente [🐮, 🐟, 🍆, 🐔, 🌽] → [🍔, 🍥, 🍟, 🍗, 🍿] → [🍟, 🍿] → 😋

> **Importante**
>
> Sin embargo es importante preguntarnos si el orden de aplicación es eficiente. Fíjate que en este caso, **hemos tenido que *'cocinar'* todos los alimentos aunque realmente Pepe solo se ha comido dos** de ellos. Este proceso hubiera sido muchísimo más eficiente si hubieramos cocinado solo aquellos que *'son vegetales'* [🐮, 🐟, 🌭, 🐔, 🌽] → [🌭, 🌽] → [🍟, 🍿] → 😋, esto es, haciendo el filtrado antes que la transformación ...
>
> **[🐮, 🐟, 🌭, 🐔, 🌽].filter(isVegetarian).map(cook).fold(🤮, eat) → 😋**
>
> **Nota:** Esto último sería posible solo si la función de filtrado ahora tuviese como conjunto **A** a los productos crudos o una **abstracción** de ambos, es decir, si se pudiera saber si un productos '**es vegetal**' independientemente de si está cocinado o no **isVegetarian(🌽) → true** y **isVegetarian(🍿) → true**

Ejemplo: Imaginemos que queremos saber el total de aprobados de una lista de notas con decimales.

```
// Select (map):      Transforma de double a double sin decimales.
//                    Puesto que Math.Round tiene varias sobrecargas, deberemos indicar
//                    mediante un cast que queremos una función del tipo A → B
//
// Where (filter):    Filtramos aquellas notas que que cumplen el predicado n >= 5d
//
// Aggregate (fold):  Vamos a llevar una cuenta en c empezando en 0d y al final del proceso
//                    convertimos la cuenta real en un valor entero. Esto último no sería
//                    necesario, si el valor inicial de la cuenta hubiera sido un literal
//                    entero, pero lo hacemos así para ves un ejemplo de uso de resultSelector.

double[] notas = [1.3, 3.4, 4.6, 5.6, 6.7, 8.7];
int aprobados = notas.Select((Func<double, double>)Math.Round)
                     .Where(n => n >= 5d)
                     .Aggregate(0d, (c, n) => c + 1, c => Convert.ToInt32(c));
Console.WriteLine($"Aprobados: {aprobados}");
```

Puesto que cosas como contar elementos en usa secuencia, acumular en una suma, buscar un máximo, etc. son casos típicos de fold, ya vienen predefinidos en C# y no tendríamos que implementarlos a través de Aggregate

```
int aprobados = notas.Select((Func<double, double>)Math.Round).Where(n => n >= 5d).Count();
```

Fíjate que es bastante similar a SQL, una posible consulta para hacer lo mismo en SQL.

```sql
SELECT COUNT(ROUND(nota)) AS aprobados FROM notas where ROUND(nota) >= 5;
```

> **Importante**
>
> Fíjate que **en este caso no podemos filtrar antes** de hacer el mapeo puesto que el resultado sería diferente. Piensa que `4.6` después del `Round` será `5` y contará como aprobado.

🚀 Ampliación opcional:

Cómo hemos dicho este patrón existirá de forma similar en otros lenguajes y seguramente sin saber la sintaxis de otros lenguajes seguramente podemos entender el código equivalente en lenguajes tan populares como:

JavaScript:

```javascript
let notas = [1.3, 3.4, 4.6, 5.6, 6.7, 8.7];
let aprobados = notas.map(Math.round)
                .filter(n => n >= 5)
                .reduce((c, n) => c + 1, 0);
document.write(`Aprobados: ${aprobados}`);
```

Python:

```python
from functools import reduce
notas = [1.3, 3.4, 4.6, 5.6, 6.7, 8.7]
aprobados = reduce(lambda c, n: c + 1,
                   filter(lambda n: n >= 5,
                          map(round, notas)),
            0)
print(f'Aprobados: {aprobados}')
```

Kotlin:

```kotlin
val notas = doubleArrayOf(
    1.3, 3.4, 4.6, 5.6, 6.7, 8.7
)
val aprobados = notas.map { Math.round(it) }
                .filter { it >= 5 }
                .fold(0) { acc, _ -> acc + 1 }
println("Aprobados: $aprobados")
```

Java:

```java
var notas = Arrays.asList(
    1.3, 3.4, 4.6, 5.6, 6.7, 8.7
);
long aprobados = notas.stream()
                .map(n -> (double) Math.round(n))
                .filter(n -> n >= 5)
                .reduce(0.0, (c, n) -> c + 1)
                .longValue();
System.out.println("Aprobados: " + aprobados);
```

Rust:

```rust
let notas = vec![1.3, 3.4, 4.6, 5.6, 6.7, 8.7];
let aprobados = notas.iter()
                .map(|&n| n.round() as i32)
                .filter(|&n| n >= 5)
                .fold(0, |c, _| c + 1);
```

Como vemos iumplementan este patrón muchos lenguajes, aunque con ligeras diferencias sintácticas. Siendo las más similares a C# las de **JavaScript** y **Kotlin**.

Otras operaciones funcionales declarativas en C#

Ya hemos visto que `Select`, `Where`, `Aggregate`, `Count` y sin duda estas operaciones como hemos mencionado nos deben *'sonar'* de SQL. Pero..., ¿Existen otras operaciones equivalentes a las que podemos encontrar en SQL?.

la respuesta es **sí** y vamos ver algunas de ellas ...

OrderBy/OrderByDescending

Ordena de manera ascendente/descendente los elementos de una secuencia en función de una clave.

Los tipos deben ser **comparables**. Si no lo son, necesitará de un objeto que implemente el interfaz `IComparer<T>` entre dos elementos del tipo de la clave por la que hemos decidido ordenar.

Distinct

Elimina valores repetidos o duplicados en una secuencia.

El tipo de datos de la secuencia deberá implementar `IEquatable` para poder comparar en igualdad los elemento de la secuencia o deberemos pasarle un objeto que implemente un interfaz de comparación.

Por ejemplo, imaginemos que queremos saber la mayor nota redondeada de la siguiente secuencia...

```
double[] notas = [ 1, 3.4, 4.3, 4.6, 4.3, 7.2, 7.6, 5.6, 8.7 ];
```

Podíamos usar un `Aggregate` para obtener el máximo o usar la función `Max` que hace la operación específica de la siguiente forma.

```
int notaMayor = notas.Aggregate(
                        double.MinValue,
                        (m, n) => n > m ? n : m,
                        n => Convert.ToInt32(Math.Round(n)));
// o también
int notaMayor = notas.Max(n => Convert.ToInt32(Math.Round(n)));

Console.WriteLine($"La nota redondeada mayor es {notaMayor}");
```

En este caso también sería bastante similar a un posible consulta en SQL.

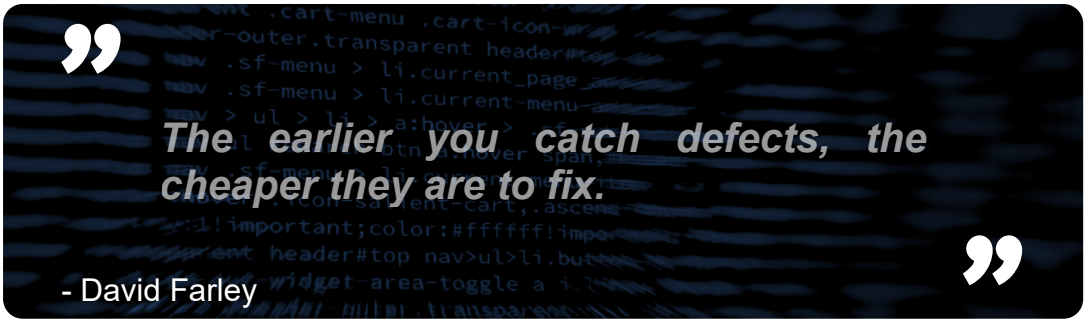
```
SELECT MAX(ROUND(nota)) AS notaMayor FROM notas;
```

Aunque más ineficiente, también podríamos usar `Distinct` y `OrderBy` de la siguiente manera ...

```
double[] notas = [ 1, 3.4, 4.3, 4.6, 4.3, 7.2, 7.6, 5.6, 8.7 ];
int notaMayor = notas.Select(n => Convert.ToInt32(Math.Round(n)))
                        // 1. Redondeamos las notas de la secuencia a su valor entero.
                        .Distinct()
                        // 2. Eliminamos de la secuencia las notas repetidas.
                        .OrderByDescending(n => n)
                        // 3. Ordenamos la secuencia de en orden descendiente, esto es,
                        // la nota mayor es la primera.
                        .First();
                        // 4. Obtenemos el primer elemento de la secuencia.
Console.WriteLine($"La nota redondeada mayor es {notaMayor}");
```

Fíjate que aunque es un poco más enrevesada, también tenemos una equivalencia en SQL.

```
SELECT DISTINCT ROUND(nota) AS notaMayor FROM notas ORDER BY notaMayor DESC LIMIT 1;
```



GroupBy

Agrupa los elementos de una secuencia según una función del selector de claves especificada y crea un valor de resultado a partir de cada grupo y su clave. Los elementos de cada grupo se proyectan utilizando una función determinada.

Existen diferentes sobrecargas posibles para usar la agrupación. Nosotros vamos a usar esta pues es la más '*intuitiva*'...

```
IEnumerable<TResult> GroupBy(
                    this IEnumerable<TSource> source,
                    Func<TSource, TKey> keySelector,
                    Func<TKey, IEnumerable<TSource>, TResult> resultSelector);
```

Veamos como interpretarla:

- `keySelector` será la función del selector de claves por las que agruparé la secuencia:
  - **Recibe**: `TSource` que es el tipo de la secuencia a procesar.
  - **Devuelve**: `TKey` tipo la clave resultado del mapeo de TSource por que que voy a agrupar.
- `resultSelector` será la función que proyecta los elemento de cada grupo con su clave:
  - **Recibe**: `Tkey` con la clave por la que agrupo y la agrupación de los objetos `TSource` para esa clave.
  - **Devuelve**: `TResult` con lo que quiero producir en la nueva secuencia resultado de la agrupación.

Bueno, hasta ahora ha sido una definición muy formal, pero vamos ha hacerlo con un ejemplo más sencillo para entenderlo. Supongamos nuestro array de notas anterior...

```
double[] notas =  [ 1, 3.4, 4.3, 4.6, 4.3, 7.2, 7.6, 5.6, 8.7 ];
```

Queremos agrupar el valor de las notas sin decimales y contabilizar el número de apariciones de las mismas.

Si estuviéramos en MySQL una posible consulta para resolverlo sería...

```
SELECT FLOOR(nota) AS notaRedondeada, count(*) AS veces
FROM notas GROUP BY notaRedondeada ORDER BY notaRedondeada;
```

El equivalente utilizando la programación funcional y el `GroupBy` de C# sería ...

```
string salida = "notaRedondeada veces\n" +
notas.GroupBy(
        nota => Math.Floor(nota),
        (nota, elementosConMismaNota) =>
                new
                {
                    NotaAgrupada = nota,
                    Veces = elementosConMismaNota.Count()
                })
     .OrderBy(datoAgrupado => datoAgrupado.NotaAgrupada)
     .Aggregate("",
                (texto, datoAgrupado) =>
                texto += $"{datoAgrupado.NotaAgrupada,-14} {datoAgrupado.Veces,-5}\n");
Console.WriteLine(salida);
```

Importante

Fíjate que aunque `string salida = ...` es una única expresión, hemos ido introduciendo saltos de línea e indentado el código para que sea legible y modificable nuestra expresión funcional. De otra manera el código sería prácticamente ilegible y difícil de modificar para un programador humano.

Si comentamos el código anterior podríamos decir que:

1. Nuestra función selectora (`Func<TSource, TKey> keySelector`) es:

   ```
   nota => Math.Floor(nota)
   ```

   Siendo `TSource` los **doubles** de la secuencia de entrada con el identificador `nota`.
   Siendo `TKey` el **double** resultado de quedarnos con la parte entera de la nota `Math.Floor(nota)`.
   Todos los elementos de la lista de entrada que produzcan la misma clave `TKey`, estarán en el mismo grupo y por tanto se agruparán en una secuencia. En nuestro caso las claves generadas serán ...

   ```
   [ 1, 3.4, 4.3, 4.6, 4.3, 7.2, 7.6, 5.6, 8.7 ] → [ 1, 3, 4, 5, 7, 8 ]
   ```

2. Nuestra función des proyección (`Func<TKey, IEnumerable<TSource>, TResult> resultSelector`) es:

```
(nota, elementosConMismaNota) => new
                {
                    NotaAgrupada = nota,
                    Veces = elementosConMismaNota.Count()
                }
```

Siendo `TKey` el **double** con la clave que recordemos es el resultado de aplicar `Floor` a la nota.
Siendo `IEnumerable<TSource>` una nueva secuencia con los elementos de la secuencia original, esto es, las notas con decimales **agrupadas a esa clave**. Esto es, los elementos que generaron esa clave.
Así pues, tendré las siguientes parejas de **(clave, agrupación)** en `(TKey, IEnumerable<TSource>)` en nuestro caso **`(nota, elementosConMismaNota)`**

```
(1, [ 1 ])
(3, [ 3.4 ])
(4, [ 4.3, 4.6, 4.3 ])
(7, [ 7.2, 7.6 ])
(5, [ 5.6 ])
(8, [ 8.7 ])
```

Siendo `TResult` la **proyección** resultante de gestionar los pares (clave, agrupación) anteriores.
En nuestro caso objetos de la '*anónimos*' con dos propiedades **`NotaAgrupada`** y **`Veces`** que tendrá la cuenta de cada secuencia de notas agrupadas con la clave.
Por tanto el resultado final será la secuencia de `TResult` siguiente...

```
[
  { NotaAgrupada = 1, Veces = 1 }, { NotaAgrupada = 3, Veces = 1 },
  { NotaAgrupada = 4, Veces = 3 }, { NotaAgrupada = 7, Veces = 2 },
  { NotaAgrupada = 5, Veces = 1 }, { NotaAgrupada = 8, Veces = 1 }
]
```

3. La función

```
.OrderBy(datoAgrupado => datoAgrupado.NotaAgrupada)
```

aplicada a la secuencia anterior de objetos anónimos producirá la misma secuencia pero ordenada por la propiedad **`NotaAgrupada`**.

4. La función

```
.Aggregate("", (texto, datoAgrupado) =>
          texto += $"{datoAgrupado.NotaAgrupada,-14} {datoAgrupado.Veces,-5}\n");
```

Genera una cadena con los pares nota sin decimales y veces que aparece alineados y separados por un salto de línea.

Ejemplo GroupBy: (Puedes descargarlo de **este enlace**)

Veamos un ejemplo algo más elaborado usando una secuencia con un tipo algo más complejo que un double. Para ello, supongamos la siguiente clase inmutable que define los datos de un empleado de una determinada empresa...

```
public enum Ciudad { Elche, Alicante };
public record Empleado(string Nombre, int Edad, Ciudad Ciudad)
{
    public override string ToString() => $"{Nombre,-9}{Edad,-3}{Ciudad}";
}
```

Ahora definimos una clase de utilidad que me devuelva una secuencia de empleados de forma *'perezosa'*...

```
public static class Empleados
{
    public static IEnumerable<Empleado> DepartamentoDeVentas
    {
        get
        {
            yield return new(Nombre: "Lola", Edad: 45, Ciudad: Ciudad.Alicante);
            yield return new(Nombre: "Pedro", Edad: 51, Ciudad: Ciudad.Alicante);
            yield return new(Nombre: "Juana", Edad: 27, Ciudad: Ciudad.Elche);
            yield return new(Nombre: "Marco", Edad: 52, Ciudad: Ciudad.Elche);
            yield return new(Nombre: "Ana", Edad: 52, Ciudad: Ciudad.Elche);
        }
    }
}
```

Queremos obtener un array con los **nombres de los empleados mayores de 40 años sin repeticiones y ordenados por nombre**.

```
string[] nombres = Empleados.DepartamentoDeVentas
                            .Where(e => e.Edad > 40)  // Filtramos por edad.
                            .Select(e => e.Nombre)    // Proyectamos la propiedad Nombre a unna nueva sec
                            .OrderBy(n => n)          // Ordenamos por nombre.
                            .Distinct()               // Eliminamos repetidos.
                            .ToArray();               // Pasamos la secuencia a array.
Console.WriteLine(string.Join(", ", nombres));
```

```
Ana, Lola, Marco, Pedro
```

Supongamos que ahora queremos obtener los **empleados mayores de 40 años agrupados por ciudad** usando `GroupBy`.

```
var empleadosXCiudad = Empleados.DepartamentoDeVentas
                                .Where(e => e.Edad > 40)
                                .GroupBy(e => e.Ciudad,
                                         (c, g) => new {Ciudad = c, Empleados = g});
```

El tipo de `empleadosXCiudad` debe ser implícito (`var`) pues la secuencia que estamos creando es de un tipo implícito (`new {Ciudad = c, Empleados = g}`) que estamos definiendo en la función de proyección del `GroupBy`.

```
Alicante:
        Lola    45 Alicante
        Pedro   51 Alicante
Elche:
        Marco   45 Elche
        Ana     52 Elche
```

Para obtener la **salida por consola dela imagen de la derecha**, podríamos hacer...

Aunque la secuencia sea de un tipo anónimo, podremos componer una cadena con la salida deseada.

```
StringBuilder salida = new StringBuilder();
foreach (var eXc in empleadosXCiudad)
{
    salida.Append($"{eXc.Ciudad}:\n");
    foreach (Empleado e in eXc.Empleados.OrderBy(e => e.Edad))
        salida.Append($"\t{e}\n");
}
Console.WriteLine(salida);
```

> **Nota**
>
> Al ser la **secuencia de un tipo anónimo**, solo podremos recorrer la secuiencia **en el mismo ámbito de uso de tipo**. Si intentáramos encapsular el la obtención de la secuencia en una función, no podríamos devolverla como tal, sino que deberíamos encapsularla en un tipo de datos o clase que la contuviera.
> Lo mismo sucedería si quisieramos encapsular el código que muestra los datos en pantalla. No sabríamos de que tipo asignarle a la secuencia que pasaríamos como parámetro.

Resultado del GroupBy en un nuevo tipo de datos

Supongamos que hubiéramos querido encapsular la obtención de `empleadosXCiudad` en una función y hacer el mismo proceso modularizado. Puesto que la secuencia es de un tipo anónimo, nos hubiéramos visto obligados ha definir una nueva clase con una propiedad `Ciudad` del tipo enumerado y otra `Empleados` como secuencia de objetos empleados por ejemplo.

> **Recuerda**
>
> A este tipo de datos o clase que definimos para obtener el resultado de una consulta y que únicamente tienen como misión transportar los datos de la consulta, se les denomina **DTO Data Transfer Object** y deben ser inmutables y tener la mínima funcionalidad posible. Por eso el modificador `record class` es el más adecuado para definirlos.

```
// Definimos el tipo.
public record class EmpleadosPorCiudadDto(Ciudad Ciudad, IEnumerable<Empleado> Empleados);

// Podemos definir un valor de retorno con un tipo concreto para la secuencia.
// En este caso hemos quitado la restricción de mayores de 40 por simplificar.
public static IEnumerable<EmpleadosPorCiudadDto> EmpleadosVentasPorCiudad() =>
Empleados.DepartamentoDeVentas
            .GroupBy(e => e.Ciudad,
                     (c, g) => new EmpleadosPorCiudadDto(Ciudad: c, Empleados: g));

// Al tener un tipo concreto, también podemos modularizar la composición de la salida.
public static string ATexto(IEnumerable<EmpleadosPorCiudadDto> empleadosXCiudad)
{
    StringBuilder salida = new StringBuilder();
    foreach (var eXc in empleadosXCiudad)
    {
        salida.Append($"{eXc.Ciudad}:\n");
        foreach (Empleado e in eXc.Empleados.OrderBy(e => e.Edad))
            salida.Append($"\t{e}\n");
    }
    return salida.ToString();
}

// Posteriormente hacer la llamada a las funciones.
Console.WriteLine(ATexto(EmpleadosVentasPorCiudad()));
```

En este caso no hemos filtrado por edad como hacíamos en el caso inicial. Pero si quisiéramos hacerlo a posteiori, podríamos hacerlo de la siguiente manera...

```
var empleadosXCiudad = EmpleadosVentasPorCiudad();

// Mapeamos al mismo DTO pero filtrando la propiedad Empleados por edad.
// Fíjate que al ser inmutable EmpleadosPorCiudadDto,
// para no tener que crear un nuevo objeto copiando propiedad a propiedad.
// Usamos el operador with para crear una nueva instancia
// con la propiedad Empleados filtrada.
var empleadosXCiudadMas40 = empleadosXCiudad.Select(
                                eXc => eXc with {
                                    Empleados = eXc.Empleados.Where(e => e.Edad > 40)
                                });

Console.WriteLine(ATexto(empleadosXCiudadMas40));
```

Resultado del GrupBy en un diccionario

Pero, ¿Habría la posibilidad de retornar la agrupación por clave sin tener que definir el tipo `EmpleadosPorCiudadDto`?. Podríamos utilizar la posibilidad de transformar cualquier secuencia a las colecciones definidas en el lenguaje.

Por ejemplo vamos a definir la misma función `EmpleadosVentasPorCiudad` pero en lugar de retornar `IEnumerable<EmpleadosPorCiudadDto>` ahora retornará `Dictionary<Ciudad, List<Empleado>>` y así no tendríamos que definir el tipo.

Es fácil e inmediato transformar una secuencia a un array o una lista.

```
IEnumerable<Empleado> secuencia = ...;

Empleado[] arrayEmpleados = secuencia.ToArray();
List<Empleado> listaEmpleados = secuencia.ToList();
```

Pero si queremos crear un diccionario, deberíamos poder indicarle de donde sacamos las claves y los valores asociados infiriendo en ambos casos los tipos parametrizados en el diccionario. Una vez más nos ayudarán la funciones de orden superior (HOF) para hacerlo en este caso podemos usar la siguiente definición de `ToDictionary`...

```
public static Dictionary<TKey, TElement>
ToDictionary<TSource, TKey, TElement>(
                this IEnumerable<TSource> source,
                Func<TSource, TKey> keySelector,
                Func<TSource, TElement> elementSelector) where TKey : notnull;
```

Donde `Func<TSource, TKey> keySelector` nos ayudará a decidir que propiedad será la clave del tipo anónimo de entrada `TSource` y `Func<TSource, TElement> elementSelector)` hará el proceso análogo para el valor asociado a la clave. Quedando la función anterior.

```
1   private static Dictionary<Ciudad, List<Empleado>> EmpleadosVentasPorCiudad() =>
    Empleados.DepartamentoDeVentas
              .GroupBy(e => e.Ciudad,
                    (c, g) => new { Ciudad = c, Empleados = g })
5             .ToDictionary(eXc => eXc.Ciudad, eXc => new List<Empleado>(eXc.Empleados));
```

Por último, queremos poder filtrar los empleados de ventas que queremos agrupar por ciudad por cualquier predicado. Una vez más podremos aplicar programación funcional y pasar la estrategia de filtrado como parámetro. (Puedes descargar el del siguiente ejemplo ejemplo en **este enlace**).

```
// En este caso no necesitamos un tipo concreto, podemos usar un diccionario.
// y mejoramos la funcionalidad anterior para que admita un filtro mediante una HOF.
private static Dictionary<Ciudad, List<Empleado>>
EmpleadosVentasPorCiudad(Func<Empleado, bool> filtroEmpleado) =>
Empleados.DepartamentoDeVentas
          .Where(filtroEmpleado)
          .GroupBy(e => e.Ciudad,
                (c, g) => new { Ciudad = c, Empleados = g })
          .ToDictionary(eXc => eXc.Ciudad, eXc => new List<Empleado>(eXc.Empleados));

private static string ATexto(Dictionary<Ciudad, List<Empleado>> empleadosXCiudad)
{
    StringBuilder salida = new StringBuilder();
    foreach (var (cuidad, empleados) in empleadosXCiudad)
    {
        salida.Append($"{cuidad}:\n");
        foreach (Empleado e in empleados.OrderBy(e => e.Edad))
            salida.Append($"\t{e}\n");
    }
    return salida.ToString();
}

static void Main()
{
    var empleadosVentasPorCiudadDeMasDe40Años =
    EmpleadosVentasPorCiudad(e => e.Edad > 40);
    Console.WriteLine(ATexto(empleadosVentasPorCiudadDeMasDe40Años));
}
```

Zip

Se trata de una operación típica en los lenguajes funcionales al permitirnos manejar el concepto de tupla y que por tanto podemos encontrar también en C# ya que también nos los permite.

Básicamente realiza una correspondencia unívoca entre los elementos de **dos** o más secuencias iterables, produciendo una **nueva secuencia de tuplas** resultado de dicha correspondencia. Obviamente, su nombre viene de la analogía de cerrar una cremallera.

Supongamos **Sec[A]** = [🐮, 🐟, 🌭, 🐔, 🌽] y **Sec[B]** = [🍔, 🍥, 🍟, 🍗, 🍿]

zip(Sec[A], Sec[B]) = [**(🐮, 🍔)**, **(🐟, 🍥)** , **(🌭, 🍟)**, **(🐔, 🍗)**, **(🌽, 🍿)**]

Ejemplo: Supongamos que tenemos dos arrays. Uno con nombres de país y otro su población en millones de habitantes. Queremos mostrar una correlación entre ambos arrays por pantalla en forma de texto. Una aproximación funcional a la solución desde C# podría ser...

```
string[] paises = ["China", "India", "Estados Unidos", "Indonesia"];
int[] poblaciones = [1391, 1364, 327, 264];

string salida  = string.Join("\n",
                 paises.Zip(poblaciones)
                       .Select(t => $"Población {t.First} {t.Second} millones."));
Console.WriteLine(salida);
```

Mostrará por pantalla...

```
Población China 1391 millones.
Población India 1364 millones.
Población Estados Unidos 327 millones.
Población Indonesia 264 millones.
```

Fíjate que la función `paises.Zip(poblaciones)` me genera una secuencia de tuplas del tipo `(string , int)` → `[("China", 1391), (India, 1364)], ... ]` y posteriormente en la función de mapeo genera una nueva secuencia de cadenas componiendo los elementos de la tupla **t**.

> **Nota**
>
> La operación inversa **Unzip** no está definida en el lenguaje porque se puede realizar fácilmente con un mapeo...

```
(string pais, int poblacion)[] datos = [
    ("China", 1391), ("India", 1364), ("Estados Unidos", 327), ("Indonesia", 264)
];
string[] paisesU = [.. datos.Select(t => t.pais)];
int[] poblacionU = [.. datos.Select(t => t.poblacion)];
Console.WriteLine(string.Join(", ", paisesU));
Console.WriteLine(string.Join(", ", poblacionU));
```

Si quisieramos hacer un `Zip` de **tres o más secuencias del mismo tamaño**, podríamos aplicar la función varias veces. En el siguiente ejemplo vamos acombinar los varires de tres vectores en una tupla de tres elementos correpondientes a coordenas en el espacio.

```
int[] v1 =  [1, 2, 3, 4, 5];
int[] v2 =  [6, 7, 8, 9, 10];
int[] v3 =  [11, 12, 13, 14, 15];

(int x, int y, int z)[] agrupacion = v1
    // Combinamos v1 con v2
    .Zip(v2, (d1, d2) => (d1, d2))
    // Combinamos las tuplas resultantes con v3
    .Zip(v3, (t12, d3) => (t12.d1, t12.d2, d3))
    .ToArray();

foreach (var (x, y, z) in agrupacion)
    Console.WriteLine($"({x}, {y}, {z})");
```

FlatMap o Aplanado

La operación de aplanado es otra de las funciones típicas de la programación funcional que vamos a encontrar en todos los lenguajes funcionales y otros como JavaScript, Java o C#. A este tipo de HOF se le conoce como **mónada** porque cumple ciertas operaciones matemáticas en la que no vamos a entrar por estar fuera del tema, pero que si has visto la **definición formal** del enlace anterior seguramente no te hayas enterado de nada si no tienes una profunda base matemática del la teoría de categorías.

Pero siguiendo la analogía que hemos usado en el tema, podemos definirla como:
flatMap = Sec[A] x (A → Sec[B]) → Sec[B]

Donde dada una **Sec[A]** le vamos a aplicar una función **A → Sec[B]** donde para cada elemento de **A** produce una secuencia de **B Sec[B]** y el resultado será una **Sec[B]** resultado de **unir** todas los **Sec[B]** producidos por cada **A**.

¿Sigue sin quedar claro?. No pasa nada, vamos a verlo a través de un ejemplo sencillo, pero esta vez en lugar de usar iconos, vamos a usar enteros y programa sencillo en C#.

En primer lugar comentaremos que la función equivalente en C# que más se aproxima a esta definición básica de **FlatMap** es **SelectMany** y en concreto la sobrecarga siguiente:

```
public static IEnumerable<B> SelectMany<A, B> (
                            this IEnumerable<TSource> source,
                            Func<A, IEnumerable<B>> selector);
```

Supongamos la siguiente de representación donde tenemos el típico array de arrays o tabla dentada. En el fondo, podemos considerarlo como una secuencia de secuencias (sub-secuencias) de enteros:

Si te fijas en el diagrama podemos ver que la operación de flat consiste en generar una nueva secuencia con datos de las sub-secuencias de entrada. El efecto es como si estuviéramos '*aplanando*' la tabla dentada.

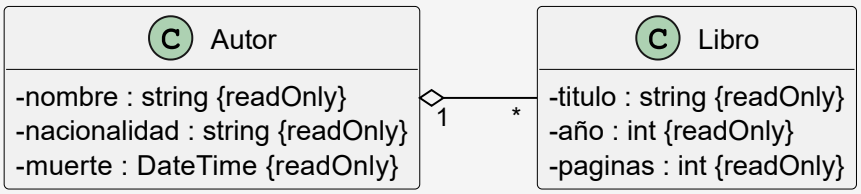
```
int[][] jagged =
[
    [11, 22],
    [34, 25, 29],
    [10],
    [17, 14, 30, 31]
];

// SelectMany = int[][] → (int[] → int[]) → int[]
int[] flat = [.. jagged.SelectMany(v => v)];
Console.WriteLine(string.Join(", ", flat));
// Mostrará por pantalla: 11, 22, 34, 25, 29, 10, 17, 14, 30, 31
```

Ejemplo SelectMany: (Puedes descargarlo desde **este enlace**).

Vemos un ejemplo más ilustrativo de este tipo de operación con tipos más elaborados.

Supongamos que la siguiente típica relación de cardinalidad uno a varios entre `Autor` y `Libro`. Donde suponemos que un autor ha escrito uno o varios libros y que modelizaremos de la siguiente manera...



El programa generará una **secuencia de autores** que mostrará al ejecutarse y que contendrá los siguientes autores:

```
Nombre: William Shakespeare
Nacionalidad: Inglesa
Muerte: 03/05/1616
Libros:
        Titulo: Macbeth                                 Año: 1623  Páginas: 128
        Titulo: La tempestad                            Año: 1611  Páginas: 160

Nombre: Miguel de Cervantes
Nacionalidad: Española
Muerte: 22/06/1616
Libros:
        Titulo: Don Quijote de la Mancha                Año: 1605  Páginas: 1376
        Titulo: La Galatea                              Año: 1585  Páginas: 664
        Titulo: Los trabajos de Persiles y Sigismunda   Año: 1617  Páginas: 888
        Titulo: Novelas ejemplares                      Año: 1613  Páginas: 1160

Nombre: Fernando de Rojas
Nacionalidad: Española
Muerte: 07/02/1541
Libros:
        Titulo: La Celestina                            Año: 1500  Páginas: 160
```

Con las funciones que hemos visto hasta el momento podríamos hacer consultas sobre los autores. Por ejemplo, para ver aquellos autores que han publicado algún libro en el siglo XVII podríamos hacer...

```
IEnumerable<Autor> autoresConLibrosPublicadosDuranteSigloXVII =
Datos.Autores.Where(a => a.Libros.Any(l => l.Año >= 1600));
```

Pero, si quisiéramos obtener los libros publicados en el siglo XVII a partir de la secuencia de autores. Ahora tendríamos un caso de uso de `SelectMany` (**Flat Map**):

```
// IEnumerable<Autor> x (Autor → IEnumerable<Libro>) → IEnumerable<Libro>
IEnumerable<Libro> librosSigloXVII =
Datos.Autores.SelectMany(a => a.Libros.Where(l => l.Año is > 1600 and < 1701));
```

Incluso podríamos **generar una correlación** entre `Libros` y `Autor` proyectando cada libro que cumple la condición a una nueva **secuencia de objetos anónimos**.

```
// IEnumerable<Autor> → (Autor → IEnumerable<'a>) → IEnumerable<'a>
// Donde 'a es un tipo anónimo que tiene los campos solicitados de la proyección.
var librosDeMenosDeMilPaginas = Datos.Autores.SelectMany(
                                        a => a.Libros.Where(l => l.Paginas < 1000)
                                                     .Select(l => new
                                                     {
                                                         Libro = l.Titulo,
                                                         Autor = a.Nombre,
                                                         Páginas = l.Paginas
                                                     }));
```

> **Aviso**
>
> Fíjate que en esta sentencia **hemos indentado el código para que sea legible**. Aunque es una única expresión funcional, el código es mucho más claro y modificable de esta manera. En una línea sería prácticamente ilegible y sería fácil cometer errores al modificarlo. Ademásm, `librosDeMenosDeMilPaginas` no puede tener un tipo explícito pues es un tipo anónimo. Por tanto debemos usar `var` para que el compilador infiera el tipo y deberíamos usar la secuencia en el mismo ámbito donde se define no pudiendo encapsularla en una función.

Por ejemplo, si quisiésemos obtener esta vista libro, autor y número de páginas y retornarla desde una función, deberíamos definir un nuevo tipo de datos **DTO** para encapsular la información.

```
public record class LibroDto(string Libro, string Autor, int Paginas)
{
    public override string ToString() => $"Libro: {Libro}\nAutor: {Autor}\nPáginas: {Paginas}";
}

public static IEnumerable<LibroDto> LibrosDeMenosdeMilPaginas() =>
Datos.Autores.SelectMany(
            a => a.Libros.Where(l => l.Paginas < 1000)
                         .Select(l => new LibroDto(Libro: l.Titulo, Autor: a.Nombre, Paginas: l.Paginas))
```

Posteriormente si ejecutamos ...

```
Console.WriteLine("\n\n");
IEnumerable<LibroDto> librosDeMenosDeMilPaginasDto = LibrosDeMenosdeMilPaginas();
Console.WriteLine(string.Join("\n\n", librosDeMenosDeMilPaginasDto));
```

Generando la siguiente salida por consola...

```
Libro: Macbeth
Autor: William Shakespeare
Páginas: 128

Libro: La tempestad
Autor: William Shakespeare
Páginas: 160

Libro: La Galatea
Autor: Miguel de Cervantes
Páginas: 664

Libro: Los trabajos de Persiles y Sigismunda
Autor: Miguel de Cervantes
Páginas: 888

Libro: La Celestina
Autor: Fernando de Rojas
Páginas: 160
```

🎓 **Caso de estudio:** (Puedes descargarlo desde **este enlace**).

Vamos a realizar varias consultas sobre el código del ejemplo anterior donde tenemos libros por autor. **Intenta pensarlas sin mirar la propuesta de solución.**

1. **Nombres** de autores que **tienen más de un libro** ordenados alfabéticamente.

```
Console.WriteLine("Consulta 1: Nombres de autores con más de un libro ordenados.");
IEnumerable<string> nombreAutoresConMasDeUnLibro = Datos.Autores
                                                        .Where(a => a.Libros.Count() > 1)
                                                        .Select(a => a.Nombre)
                                                        .OrderBy(n => n);
Console.WriteLine(string.Join("\n", nombreAutoresConMasDeUnLibro) + "\n");
```

```
Consulta 1: Nombres de autores con más de un libro ordenados.
Miguel de Cervantes
William Shakespeare
```

2. Total de **libros** escritos por escritores de **nacionalidad Española**.

```
Console.WriteLine("Consulta 2: Total de libros escritos por escritores Españoles.");
var totalLibros = Datos.Autores
                       .Where(a => a.Nacionalidad == "Española")
                       .SelectMany(a => a.Libros)
                       .Count();
Console.WriteLine($"Hay {totalLibros} libros escritos por españoles\n");
```

```
Consulta 2: Total de libros escritos por escritores Españoles.
Hay 5 libros escritos por españoles
```

3. **Nombre** y **año muerte** de Autores **agrupados por siglo** en el que murioeron.

```
Console.WriteLine("Consulta 3: Nombre y año muerte de Autores agrupados por siglo en el que murioeron
var autoresFallecidosPorSiglos = Datos.Autores
                                        .Select(a => new { Autor = a.Nombre, AñoMuerte = a.Muerte.Ye
                                        .OrderBy(a => a.AñoMuerte)
                                        .GroupBy(a => a.AñoMuerte / 100 + 1,
                                                (siglo, autores) => new
                                                {
                                                    Siglo = siglo,
                                                    Autores = autores
                                                });
StringBuilder salida = new();
foreach (var autoresXSiglo in autoresFallecidosPorSiglos)
{
    salida.AppendLine($"Siglo {autoresXSiglo.Siglo}:");
    foreach (var autor in autoresXSiglo.Autores)
        salida.AppendLine($"\t{autor.Autor} fallecido en {autor.AñoMuerte}");
}
Console.WriteLine(salida);
```

```
Consulta 3: Nombre y año muerte de Autores agrupados por siglo en el que murioeron.
Siglo 16:
        Fernando de Rojas fallecido en 1541
Siglo 17:
        William Shakespeare fallecido en 1616
        Miguel de Cervantes fallecido en 1616
```

4. Total de **páginas** escritas por **William Shakespeare**.

```
Console.WriteLine("Consulta 4: Número total de páginas escritas por William Shakespeare.");
var totalPaginas = Datos.Autores.Where(a => a.Nombre == "William Shakespeare")
                                .SelectMany(a => a.Libros.Select(l => l.Paginas)).Sum();
Console.WriteLine($"William Shakespeare escribió {totalPaginas} páginas.");
```

```
Consulta 4: Número total de páginas escritas por William Shakespeare.
William Shakespeare escribió 288 páginas.
```

Anexo I: Concepto de Recursión de funciones o Recursividad

Definiciones

Llamada recursiva: Se produce cuando un método se llama a si mismo para realizar el proceso u obtener el valor de retorno.
Recursividad: La capacidad de un módulo de llamarse a si mismo.
Algoritmo recursivo: Un algoritmo resuelto mediante **recursividad**.

> **Importante**
>
> Cualquier **algoritmo recursivo** tiene un equivalente **iterativo**.

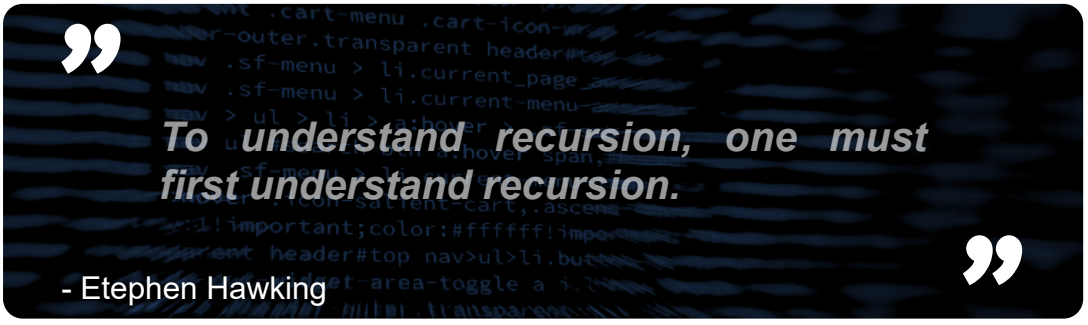
✅ Ventajas

Existen problemas cuya **solución natural es claramente recursiva** con lo que el algoritmo resulta **muy simple y claro frente** a la solución iterativa. Por ejemplo ...

- El recorrido de colecciones tipo **árbol** o **grafo**.
- Algoritmos que generan árboles en la búsqueda de una solución como el **Backtracking**.

❌ Desventajas

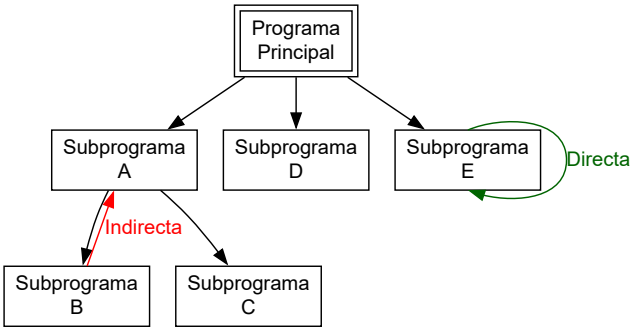
- Generalmente es **menos eficiente** que la solución iterativa. Aunque se puede aliviar con una técnica denominada **memoization**.
- Consume muchos más recursos.
- Puede llevar al **desbordamiento de la pila** de llamadas o (**Stack Overflow**).



Tipos de recursividad

Recursividad directa: se produce cuando un subprograma o módulo se llama a si mismo.

Recursividad indirecta: se produce cuando un subprograma o módulo **A** llama a otro **B** que acaba llamando de nuevo a **A**.



> **Peligro**: La **recursividad indirecta**, no deberíamos usarla **en ningún caso**.

Diseño de un algoritmo recursivo

Necesitaremos definir o buscar dos cosas:

1. **El caso general**
   - Deberemos subdividir el problema en **versiones menores de si mismo**.
     - Pueden darse varios casos generales.
     - Todos **tenderán** a un determinado caso base.
   - Deberemos confiar en que la llamada recursiva va a hacer su trabajo y va a solucionar o devolver parte del problema.
2. **Al menos una caso base**
   - Condición de salida de la recursividad o llamada recursiva, donde el caso general ya no se puede subdividir más y donde dejamos de hacer llamadas recursivas.
   - Pueden darse varios casos base.
   - Si **no existe el caso base**, no saldríamos de la recursividad y se produciría un **Stack Overflow** 💀.

Punto de vista desde el lenguaje CSharp

- Tendremos una **pila de llamadas** al mismo método.
- En cada llamada se apilan o *'guardan'* y no se modifican los objetos locales y parámetros del método llamador.
- Al recuperar el control, el valor de los objetos apilados es recuperado.
- Es interesante usar el depurador para ver como funcionan este tipo de programas.

Ejemplo:

Veámoslo a través de un problema que ya hemos solucionado de forma iterativa, pero que tiene una solución matemática natural recursiva.

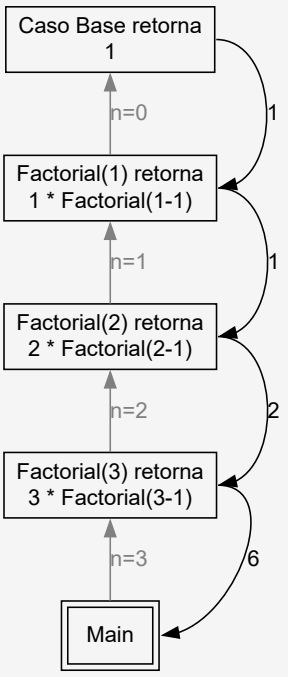
Vamos a implementar una **función recursiva para calcular el factorial de un número entero**. Recordemos que el factorial se simboliza como $n!$, se lee como '*n factorial*', y la definición es: $n! = n \times (n-1) \times (n-2) \times ... \times 1$. Además, no se puede calcular el factorial de números negativos, y el factorial de 0 es 1, de modo que una función bien hecha para cálculo de factoriales debería incluir un control para esos casos.

Pasos:

1. La función debería tener un interfaz descriptivo, tal y como lo tiene la iterativa, esto es: `ulong Factorial(ulong n)`
2. Identificaremos el caso base, en este caso es **si n = 0 → !n = 1**
3. Identificaremos el caso general, esto es, cómo descomponer el problema de tal manera que tienda al caso base. En este caso será **n! = n X (n - 1)!**

```
static ulong Factorial(ulong n)
{
    ulong valor;

    if (n == 0)
        valor = 1;                      // Caso Base
    else
        valor = n * Factorial(n - 1);   // Caso General
    return valor;
}
```



La variable local `valor` y el parámetro formal `n` será diferente en cada llamada y se liberarán de memoria en cuanto se **desapilen** la llamadas a la función.

Si ejecutásemos el siguiente código:

```
static void Main()
{
    Console.WriteLine(Factorial(3));
}
```

Usando funciones Lambda (λ)

Para expresar la función factorial anterior con expresiones lambda haríamos algo cómo ...

```
Func<int, int> factorial = n => (n > 0) ? n * factorial(n - 1) : 1;
```

Pero como aún no hemos completado la instrucción donde **definimos** el identificador del delegado factorial **nos dice que aún no existe al usarlo**.

Existen varias soluciones al problema '*elegantes*' desde el punto de vista **funcional y matemático**. Pero una solución muy simple sería definir primero el **identificador** que referencia al delegado asignándole `default`.

Cómo ya está definido el identificador y es una referencia a un delegado. Podremos **reasignarlo** haciendo una clausura dentro del mismo la referencia.

```
Func<int, int> factorial = default;
factorial = n => (n > 0) ? n * factorial(n - 1) : 1;
Console.WriteLine(factorial(3));
```

🚀 **Ampliación opcional:**

Fíjate que la implementación en otros lenguajes es similar, pero en los lenguajes dinámicos como JavaScript o Python no es necesario definir el tipo antes como sucede en C# o Kotlin.

JavaScript:

```
let factorial = n => (n > 0) ? n * factorial(n - 1) : 1;
console.log(factorial(3))
```

Python:

```
factorial = lambda n : n * factorial(n - 1) if (n > 0) else 1
print(factorial(3))
```

Kotlin:

```
var factorial: (Int) -> Int
factorial = { n -> if (n > 0) n * factorial(n - 1) else 1 }
println(factorial(3))
```

Ejemplo 1:

Vamos ha hacer una **función recursiva que calcule la potencia de un numero**.

Interfaz: `double Pow(double b, int e)` ... Debería ser igual que el de la iterativa.
Caso Base: $b^0 \rightarrow 1$
Caso General: $b^e \rightarrow b \times b^{e-1}$

Propuesta de solución:

```
static double Pow(double b, int e)
{
    double p;
    if (e <= 0)
        // Caso Base
        p = 1d;
    else
        // Caso General
        p = b * Pow(b, e - 1);
    return p;
}
```

Propuesta con λ:

```
Func<double, int, double> pow = default;
pow = (b, e) => e > 0 ? b * pow(b, e - 1) : 1d;
```

Ejemplo 2:

Vamos a implementar una función que dado un número decimal de entrada, me devuelva una cadena con su representación en binario (base 2). En este ejemplo es un poco más difícil de ver el caso base y general.

Interfaz: `string Binario(int valorDecimal)`
*Caso Base**: Habrá dos casos base que son los valores que no necesitan conversión por ser menores que 2 que que són los dos posibles cocientes que concatenaré en primer lugar ...

1. valorDecimal = 0 → cadenaBinaria = "0"
2. valorDecimal = 1 → cadenaBinaria = "1"
   Caso General: Binario(valorDecimal / 2) + "Resto de dividir valorDecimal por 2"
   Como he concatenar los restos de del último al primero, por eso voy concatenando los restos por la derecha.

Propuesta de solución:

```
static string Binario(int valorDecimal)
{
    string cadenaBinaria;
    switch (valorDecimal)
    {
        case 0:
            cadenaBinaria = "0";
            break;
        case 1:
            cadenaBinaria = "1";
            break;
        default:
            cadenaBinaria =
            Binario(valorDecimal / 2)
            + $"{valorDecimal % 2}";
            break;
    }
    return cadenaBinaria;
}
```

Propuesta con λ:

```
Func<int, string> binario = default;
binario = d => d switch
{
    0 => "0",
    1 => "1",
    _ => binario(d / 2) + $"{d % 2}",
};
Console.WriteLine(binario(44)); // Mostrará "101100"

// Algo más ofuscada podría ser la siguiente expresión.
Func<int, string> binario = default;
binario = d => d > 1
          ? binario(d / 2) + $"{d % 2}"
          : (d == 0 ? "0" : "1");
```

Ejemplo 3:

Dada una cadena de entrada, implementar una función recursiva que me devuelva su inversa.

Interfaz: `string Invierte(string t)`
Caso Base: Si la longitud de la cadena de entrada es 1 su inversa es ella misma.
Caso General: Invierte(t menos la primera letra) + primera letra de t

Propuesta de solución:

```
static string Invierte(string t)
{
    string invertida;
    if (t.Length > 1)
        invertida =
        Invierte(t.Substring(1, t.Length - 1))
        + t[0];
    else
        invertida = t;
    return invertida;
}
```

Propuesta con λ:

```
Func<string, string> invierte = default;
invierte = t => t.Length > 1
           ? invierte(t[1..]) + t[0]
           : t;
```

Resumen

Existen problemas típicos mucho más complejos a resolver mediante recursividad pero quedan fuera de lo que pretende este tema que es una mera introducción al concepto.